VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÝ EDITOR PRO BLOKOVÝ VSTUP NUMERICKÉHO INTEGRÁTORU V .NET

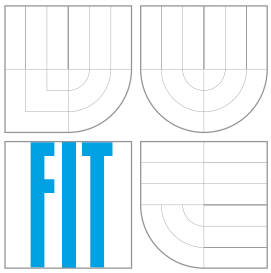
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN KUČERA

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÝ EDITOR PRO BLOKOVÝ VSTUP NUMERICKÉHO INTEGRÁTORU V .NET

NUMERICAL INTEGRATION .NET GRAPHICAL EDITOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN KUČERA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VÁCLAV ŠÁTEK, Ph.D.

BRNO 2012

Abstrakt

Tato bakalářská práce se zabývá numerickým řešením soustav diferenciálních rovnic prvního řádu s počátečními podmínkami vybranými jednokrokovými metodami. Dále pak návrhem, implementací a testováním aplikace, která je schopna užitím zmíněných metod řešit úkoly zadané blokovými schématy. První část je věnována úvodu do problematiky, obsahuje příklad výpočtu několika kroků vybranými metodami a srovnává dosažené výsledky. Podstatná část zprávy se zabývá podobou a funkcí jednotlivých bloků použitých v editoru blokových schémat a procesu transformace vstupních dat na data použitelná pro vytvoření simulace. Předposlední kapitola ukazuje jaké přesnosti dosahuje výsledná aplikace. Je zde předvedeno řešení několika jednoduchých příkladů z praxe, nechybí srovnání s jinými simulačními systémy.

Abstract

This thesis deals with the numerical solution of systems of first-order differential equations with initial conditions, using chosen one-step methods. It describes the design, implementation and testing of the application that is able to use these methods to solve the tasks entered in form of block schemes. The first part is devoted to an introduction of the topic, it provides an example of calculation of several steps using chosen methods and compares the results. A substantial part of the report deals with form and function of each block used in the block schemes editor and the transformation process of input data to data usable for creating simulation. The penultimate chapter demonstrates the accuracy of the final application. It shows solutions of a few simple examples of practical usage. This chapter also includes comparison with other simulation systems.

Klíčová slova

diferenciální rovnice, grafické uživatelské rozhraní, blokové schéma, ODR, numerické metody, Simulink, OpenModelica

Keywords

differential equation, graphical user interface, block scheme, ODE, numerical methods, Simulink, OpenModelica

Citace

Martin Kučera: Grafický editor pro blokový vstup numerického integrátoru v .NET, bakalářská práce, Brno, FIT VUT v Brně, 2012

Grafický editor pro blokový vstup numerického integrátoru v .NET

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Václava Šátka, Ph.D. Další informace mi poskytl Ing. Václav Vopěnka a Ing. Pavla Sehnalová, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Kučera
14. května 2012

Poděkování

Zde bych chtěl poděkovat mému vedoucímu, kterým byl Ing. Václav Šátek, Ph.D., za trpělivost, ochotu a vždy příjemnou atmosféru. V rámci své bakalářské práce se mnou spolupracoval na návrhu a implementaci výsledné aplikace Jiří Kopecký, kterému tímto velmi děkuji. Děkuji také Ing. Václavu Vopěnkovi a Ing. Pavle Sehnalové, Ph.D. za užitečné rady.

Obsah

Kapitola 1

Úvod

Práce se zabývá tématem numerického výpočtu soustav diferenciálních rovnic (prvního řádu s počátečními podmínkami). Diferenciální rovnice se používají pro popis jevů z reálného světa. Soustavu rovnic popisující chování jevu označujeme jako matematický model. Z matematického modelu jsme schopni vytvořit simulační model, nad kterým můžeme provádět experimenty. Jsme schopni provést velké množství různých testů a experimentů nad simulačním modelem pouze za zlomek zdrojů, které bychom museli jinak vynaložit při testování a experimentování na fyzickém systému.

Diferenciální rovnice se řeší dvěma způsoby. Prvním z nich je analytické řešení, které je velmi přesné a výsledky lze získávat rychle. Bohužel ne všechny rovnice (a jejich soustavy) lze řešit analyticky. Velká část problémů z praxe řešit analyticky nelze. Proto existuje druhý způsob a tím je numerické řešení, které využívá aproximaci zadané funkce. Oproti analytickému je časově náročnější, navíc hodně záleží na dalších faktorech. Jedním z nich je např. výběr metody – ne všechny metody jsou vhodné pro všechny problémy.

Práce je zaměřena na simulaci obvodů. Budeme používat tzv. spojitou simulaci – čas v simulaci se opakovaně zvyšuje po malých krocích a v každém kroku se přepočítají všechny děje, které v systému probíhají.

1.1 Cíle

Práce si klade za cíl vyvinout program, který v budoucnu ve výuce předmětů Teorie obvodů a Vysoce náročné výpočty nahradí TKSL/386. Především proto, že na moderních 64-bitových operačních systémech Windows nelze tento starší program spustit (ne bez nějaké formy emulace, např. DOSBox). Také uživatelské prostředí (textový režim) je již zastaralé a pro studenty neintuitivní.

Úkolem je implementovat překladač a aplikaci pro výpočet soustav obyčejných diferenciálních rovnic prvního řádu s počátečními podmínkami v MS .NET. Díky využití platformy .NET získáme podporu dalších platforem, mimo jiné i systémů POSIX (GUI napsané ve WPF je bohužel nepřenositelné, ale konzolová aplikace je testovaná a plně funkční pod GNU/Linux). Z podporovaných jazyků bude zvolen `C#`. Jedná se o moderní jazyk, který vychází především z jazyků `C++` a `Java`. Více o .NET a `C#` si můžete přečíst v [12].

Dále je nutné navrhnout datové struktury, jednotlivé vrstvy aplikace a jazyk. Jazyk nebude sloužit jen pro textový vstup, ale bude využit i pro návrh mezikódu. Mezikód bude výstupem front-endu (např. textového vstupu) a bude sloužit jako předpis pro vytvoření simulační třídy (zatím jediný back-end bude numerický simulátor). I blokový vstup bude

kompilován do stejného mezikódu jako textový. Tento mezikód bude pravděpodobně svým uspořádáním hodně blízký právě textovému vstupu (např. seznamy diferenciálních rovnic, seznamy proměnných atp.).

Výsledný program musí umět více metod řešení – základní testovací numerické metody budou explicitní Eulerova a Runge-Kutta čtvrtého řádu.

Vstupními daty aplikace bude blokové schéma – tzn. implementace grafického rozhraní (editoru blokového schématu) je jedním z hlavních úkolů.

1.2 Kapitoly

V kapitole 2 nastíním teoretické pozadí, především popíšu numerické řešení diferenciálních rovnic různými metodami.

V následující kapitole 3 vysvětlím v krátkosti z čeho jsme vycházeli při návrhu vlastností výpočetního systému a textového vstupu. Právě z podoby textového vstupu jsme postupně navrhli vnitřní reprezentaci kódu (mezikód), která je společná pro všechny vstupy.

Kapitola 4 se zabývá implementací, bude obsahovat popis jednotlivých bloků, několik příkladů blokových schémat a popis blokového editoru.

Ukázku použití aplikace, srovnání grafického vstupu a výstupu s jiným výpočetním systémem naleznete v kapitole 5.

Kapitola 2

Diferenciální rovnice

Obyčejné diferenciální rovnice se velmi často užívají při řešení problémů z oblasti fyziky, chemie a technických oborů.

Jde o takové matematická rovnice, ve kterých se vyskytuje vztah mezi funkcí jedné proměnné a jejími derivacemi (nejvyšší derivace určuje řád diferenciální rovnice). Jednoduchý příklad diferenciální rovnice:

$$\frac{\mathsf{d}u(t)}{\mathsf{d}t} = u(t) \tag{2.1}$$

Řešením této rovnice je funkce:

$$u(t) = c \cdot e^t \tag{2.2}$$

ve které c je libovolná konstanta (tato konstanta se vypočítá ze zadané počáteční podmínky, většinou v $t = 0$).

Širší úvod do problematiky si můžete přečíst např. v [2].

2.1 Analytické řešení

Analyticky řešit umíme jen určité typy diferenciálních rovnic. V praktických úlohách se většinou vyskytují rovnice, které analyticky lze řešit velmi obtížně (pracně), nebo je vyřešit analyticky nelze vůbec. Detailnější informace můžete nalézt v [1].

2.2 Numerické řešení

Většinu diferenciálních rovnic nelze řešit analyticky (nebo je lze řešit, ale je to příliš náročné), proto se používají numerické metody pro řešení diferenciálních rovnic (a jejich soustav). Tato řešení jsou pouze přibližná, ale při vhodně zvolených parametrech dostaneme dostatečně přesné (tudíž použitelné) výsledky. Výstupem těchto metod není funkce, jako tomu bylo u analytického řešení, ale přibližné hodnoty řešení v konečném počtu bodů. V této kapitole představím několik nejznámějších metod.

2.2.1 Vlastnosti numerických metod

Explicitní metody

Explicitní metoda je taková metoda, která v předpisu k výpočtu dalšího kroku nemá neznámou nového kroku (y_{i+1}) i na pravé straně.

Příklad explicitní metody (Adams-Bashforth čtvrtého řádu)[5]:

$$y_{i+1} = y_i + \frac{h}{24}(55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}) \tag{2.3}$$

Implicitní metody

Jsou na výpočet složitější než explicitní, nestačí pouze dosadit a vypočítat. Jedná se o takové metody, které ve svých rovnicích pro výpočet y_{i+1} obsahují f s parametrem y_{i+1}. Mají obecně lepší oblast stability, ale bývají náročnější na řešení (musíme řešit soustavy nelineárních rovnic vhodnou iterační metodou).

Implicitní numerické metody se používají např. při řešení tuhých (stiff) systémů diferenciálních rovnic, které vyžadují od metody velkou oblast stability.

Příklad implicitní metody (Adams-Moulton čtvrtého řádu)[5]:

$$y_{i+1} = y_i + \frac{h}{24}(9\underline{f_{i+1}} + 19f_i - 5f_{i-1} + f_{i-2}) \tag{2.4}$$

Chyby

Když je $\hat{x}$ přesná hodnota a x je její aproximace, pak jejich rozdíl je **absolutní chyba** aproximace (někdy také celková diskretizační chyba).

$$E = \hat{x} - x \tag{2.5}$$

Často je užitečnější používat relativní chybu aproximace

$$RE(x) = \frac{\hat{x} - x}{x} = \frac{E(x)}{x} \tag{2.6}$$

která se obvykle vyjadřuje v procentech.

Lokální diskretizační chybou rozumíme takovou chybu, které jsme se dopustili při jednom kroku dané metody (když předpokládáme, že všechny hodnoty potřebné k výpočtu tohoto kroku jsou přesné). Celková (globální) diskretizační chyba je nakupením lokálních diskretizačních chyb, někdy se značí také jako e_i. Je proto důležité, aby při užití dané metody nedocházelo ke katastrofální akumulaci lokálních diskretizačních chyb. Pokud se tak neděje, tak o takové metodě řekneme, že je **stabilní**.

Celková diskretizační chyba klesá, když se volí menší krok (u vhodně zvolených metod pro danou úlohu). Při příliš malém kroku (nebo u určitých typů úloh) ale vzrůstá celková **zaokrouhlovací chyba** (důvodem vzniku této chyby je omezený prostor pro ukládání čísel v paměti počítače).

Více o chybách se můžete dočíst v [3].

Řád

Jak je uvedeno v [6] – řád metody je největší přirozené číslo p takové, pro které platí odhad lokální diskretizační chyby následovně:

$$d_i = O(h_i^{p+1}) \tag{2.7}$$

2.2.2 Metoda Taylorova rozvoje

Mějme funkci y, které je řešením rovnice $y' = f(t, y)$. Předpokládáme, že funkce f je nekonečně diferencovatelná a funkce y má derivace všech vyšších řádů

$$y''(t) = \frac{\mathrm{d}}{\mathrm{d}t} f(t,y) = f_t(t,y) + f_y(t,y)y' = f_t(t,y) + f_y(t,y)f(t,y) \tag{2.8}$$

$$\begin{aligned} y'''(t) &= \frac{\mathrm{d}^2}{\mathrm{d}t^2} f(t,y) = f_{tt}(t,y) + 2f_{ty}(t,y)f(t,y) + f_t(t,y)f_y(t,y) + \\ &+ [f_y(t,y)]^2 f(t,y) + f_{yy}(t,y)[f(t,y)]^2 \end{aligned} \tag{2.9}$$

kde $f_t(t, y)$ značí parciální derivaci $f(t, y)$ podle t (zavedeno kvůli zjednodušení zápisu). Ze stejného důvodu budeme zapisovat úplné derivace pomocí horních indexů následovně

$$y^{(p+1)}(t) = f^{(p)}(t, y), p = 1, 2, ... \tag{2.10}$$

Užitím Taylorova vzorce a výše zmíněného značení dostaneme přírůstek řešení y na intervalu $< t, t + h >$ takto

$$\begin{aligned} y(t+h) &= y(t) + hf(t,y) + \frac{1}{2}h^2 f'(t,y) + \dots \\ &\quad \dots + \frac{1}{p!}h^p f^{(p-1)}(x,y) + O(h^{p+1}) \end{aligned} \tag{2.11}$$

Nyní jsme dostali jednokrokovou metodu řádu p. K výpočtu hodnoty se musí vyjádřit vyšší derivace v (2.11) užitím parciálních derivací (ukázkou jsou rovnice (2.8) a (2.9)).

Tato metoda není pro obecné typy rovnic (a jejich soustav) příliš vhodná, protože analytické derivování může být hodně pracné pro $p > 3$. Navíc funkce $y(t)$ musí mít derivace alespoň do řádu p.

Podrobnější informace o této metodě i s příkladem lze nalézt např. v [9].

2.2.3 Explicitní Eulerova metoda

Dosazením $p = 1$ do rovnice (2.11) dostaneme explicitní Eulerovu metodu. Tato metoda je jednoduchá metoda prvního řádu. Bohužel je také velmi nepřesná, musí se volit malý krok, abychom dostali relevantní výsledky.

Tvar:

$$y_{i+1} = y_i + h \cdot f(t_i, y_i) \tag{2.12}$$

Kde y_i je hodnota y v předchozím kroku, h je velikost kroku, $f(t_i, y_i)$ je derivace y_i v t_i a y_{i+1} je nová hodnota y v tomto kroku.

Příklad

Mějme zadání:

$$\begin{aligned} y' &= y \\ y(0) &= 1 \\ y(0.02) &= ? \end{aligned} \tag{2.13}$$

Krok zvolíme $h = 0.01$. Hodnoty t_0 a y_0 získáme pouhým dosazením.

$$t_0 = 0$$
$$y_0 = y(0) = 1$$

Dále vypočteme následující t

$$t_1 = t_0 + h = 0 + 0.01 = 0.01$$

a dosadíme do předpisu Eulerovy explicitní metody (2.12).

$$y_1 = y_0 + h \cdot f(t_0, y_0) = 1 + 0.01 \cdot 1 = 1.01$$

Další krok se vyřeší obdobně.

$$t_2 = t_1 + h = 0.01 + 0.01 = 0.02$$
$$y_2 = y_1 + h \cdot f(t_1, y_1) = 1.01 + 0.01 \cdot 1.01 = 1.0201$$

číslo kroku	t	y
0	0	1
1	0.01	1.01
2	0.02	1.0201

Tabulka 2.1: Shrnutí výsledků

2.2.4 Runge-Kuttovy metody

Jde snad o nejznámější skupinu jednokrokových metod. Obecný tvar vypadá následovně:

$$y_{i+1} = y_i + h(w_1 k_1 + ... + w_s k_s) \tag{2.14}$$

kde

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_n &= f(t_i + \alpha_n h, y_i + h \sum_{j=1}^{n-1} \beta_{nj} k_j), \quad n = 2, \ldots, s \end{aligned} \tag{2.15}$$

a w_n, α_n a β_{nj} jsou konstanty vybrané tak, aby výsledná metoda měla co nejvyšší řád.

Více o Runge-Kuttových metodách najdete např. v [15].

2.2.5 Butcherova tabulka

Pro přehledný zápis Runge-Kuttových metod se používá Butcherova tabulka.

$$\begin{array}{c|ccccc} 0 & & & & & \\ \alpha_2 & \beta_{21} & & & & \\ \alpha_3 & \beta_{31} & \beta_{31} & & & \\ \vdots & \vdots & & \ddots & & \\ \alpha_n & \beta_{n1} & \beta_{n2} & \ldots & \beta_{n,n-1} & \\ \hline & w_1 & w_2 & \ldots & w_{n-1} & w_n \end{array}$$

Tabulka 2.2: Obecná podoba

α, β a w jsou konstanty z rovnic (2.14) a (2.15).

Příklad

Explicitní Eulerova metoda má následující tabulku (srovnejte s rovnicí (2.12))

0	
	1

Tabulka 2.3: Butcherova tabulka Eulerovy metody

2.2.6 Heunova metoda

Heunova metoda je modifikací Eulerovy metody. Na rozdíl od ní ale počítá y_{i+1}(hodnotu v dalším kroku) pomocí dvou bodů, ne jen jednoho.

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + h, y_i + hk_1) \qquad (2.16) \\ y_{i+1} &= y_i + \tfrac{1}{2}h(k_1 + k_2) \qquad (2.17) \end{aligned}$$

Heunova metoda je explicitní metoda druhého řádu. Rovnice (2.16) a (2.17) jsou předpisem této metody, zápis ve zkrácené podobě pomocí Butcherovy tabulky vidíme v tabulce 2.4. Čerpal jsem z [4].

0		
1	1	
	1/2	1/2

Tabulka 2.4: Butcherova tabulka Heunovy metody

Příklad

Použijeme stejný příklad (2.13) a krok jako u Eulerovy metody – $h = 0.01$.

Hodnoty v počátku získáme dosazením:

$$\begin{aligned} t_0 &= 0 \\ y_0 &= y(0) = 1 \end{aligned}$$

Dále vypočteme následující t

$$t_1 = t_0 + h = 0 + 0.01 = 0.01$$

a dosadíme do předpisu (2.16) a následně do (2.17):

$$\begin{aligned} k_1 &= f(t_0, y_0) = 1 \\ k_2 &= f(t_0 + h, y_0 + hk_1) = 1 + 0.01 \cdot 1 = 1.01 \\ y_1 &= y_0 + \frac{h}{2}(1 + 1.01) = 1 + \frac{0.01 \cdot 2.01}{2} = 1 + \frac{0.0201}{2} = 1.01005 \end{aligned}$$

Nyní vše zopakujeme pro další krok:

$$\begin{aligned} k_1 &= f(t_1, y_1) = 1.01005 \\ k_2 &= f(t_1 + h, y_1 + hk_1) = 1.01005 + 0.01 \cdot 1.01005 = 1.0201505 \\ y_2 &= y_1 + \frac{h}{2}(1.01005 + 1.0201505) = 1.01005 + \frac{h}{2}(1.01005 + 1.0201505) = \\ &= 1.01005 + \frac{0.01 \cdot 2.0302005}{2} = 1.01005 + 1.01510025 = 1.0202010025 \end{aligned}$$

číslo kroku	t	y
0	0	1
1	0.01	1.01005
2	0.02	1.0202010025

Tabulka 2.5: Shrnutí výsledků

2.2.7 Runge-Kutta 3

Metoda třetího řádu z rodiny Runge-Kuttových metod. Na rozdíl od doposud uvedených metod se tato již běžně používá v praxi při řešení problémů. Má následující předpis

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + \tfrac{1}{3}h, y_i + \tfrac{1}{3}hk_1) \\ k_3 &= f(t_i + \tfrac{2}{3}h, y_i + \tfrac{2}{3}hk_2) \qquad (2.18) \\ y_{i+1} &= y_i + \tfrac{1}{4}h(k_1 + 3k_3) \qquad (2.19) \end{aligned}$$

Informace jsem získal z [14].

0			
1/3	1/3		
2/3	0	2/3	
	1/4	0	3/4

Tabulka 2.6: Butcherova tabulka metody

Příklad

Řešme stejný příklad jako u Eulerovy metody - (2.13). Hodnoty t_0 a y_0 získáme dosazením

$$\begin{aligned} t_0 &= 0 \\ y_0 &= y(0) = 1 \end{aligned}$$

vypočteme následující t

$$t_1 = t_0 + h = 0 + 0.01 = 0.01$$

a dosadíme do předpisu (2.18) a následně do (2.19):

$$\begin{aligned} k_1 &= f(t_0, y_0) = 1 \\ k_2 &= f(t_0 + \tfrac{1}{3}h, y_0 + \tfrac{1}{3}hk_1) = 1 + \frac{0.01 \cdot 1}{3} \doteq 1.0033333333 \\ k_3 &= f(t_0 + \tfrac{2}{3}h, y_0 + \tfrac{2}{3}hk_2) = 1 + \frac{2 \cdot 0.01 \cdot 1.0033333333}{3} \doteq 1.0066888889 \\ y_1 &= y_0 + \tfrac{1}{4}h(k_1 + 3k_3) = 1 + \frac{0.01(1 + 3 \cdot 1.0066888889)}{4} \doteq 1.0100501667 \end{aligned}$$

Obdobně vypočítáme další krok

$$
\begin{aligned}
k_1 &= f(t_1, y_1) = 1.0100501667 \\
k_2 &= f(t_1 + \tfrac{1}{3}h, y_1 + \tfrac{1}{3}hk_1) = 1.0100501667 + \frac{0.01 \cdot 1.0100501667}{3} \doteq 1.0134170006 \\
k_3 &= f(t_1 + \tfrac{2}{3}h, y_1 + \tfrac{2}{3}hk_2) = 1.0100501667 + \frac{2 \cdot 0.01 \cdot 1.0134170006}{3} \doteq 1.0168062800 \\
y_2 &= y_1 + \tfrac{1}{4}h(k_1 + 3k_3) = 1.0100501667 + \\
&+ \frac{0.01 \cdot (1.0100501667 + 3 \cdot 1.0168062800)}{4} \doteq 1.0202013392
\end{aligned}
$$

číslo kroku	t	y
0	0	1
1	0.01	1.0100501667
2	0.02	1.0202013392

Tabulka 2.7: Shrnutí výsledků

2.2.8 Runge-Kutta 4

Nejvíce používaná z této skupiny je (explicitní) Runge-Kutta 4. řádu. Když se mluví o Rungově-Kuttově metodě, většinou má řečník na mysli právě tuto.

$$
\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f(t_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_1) \\
k_3 &= f(t_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_2) \\
k_4 &= f(t_i + h, y_i + hk_3) && (2.20) \\
y_{i+1} &= y_i + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) && (2.21)
\end{aligned}
$$

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Tabulka 2.8: Butcherova tabulka metody

Příklad

Mějme stejné zadání jako v případě Eulerovy metody - (2.13). Dosadíme počáteční podmínky, získáme

$$t_0 = 0$$

$$y_0 = 1$$

Vypočteme si pomocné proměnné k_1 až k_4 (užijeme rovnice (2.20))

$$\begin{aligned} k_1 &= f(t_0, y_0) = 1 \\ k_2 &= f(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}hk_1) = 1 + \frac{0.01 \cdot 1}{2} = 1.005 \\ k_3 &= f(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}hk_2) = 1 + \frac{0.01 \cdot 1.005}{2} = 1.005025 \\ k_4 &= f(t_0 + h, y_0 + hk_3) = 1 + 0.01 \cdot 1.005025 = 1.01005025 \end{aligned}$$

a dosadíme do vzorce (2.21)

$$\begin{aligned} y_1 &= y_0 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) = \\ &= 1 + \frac{0.01 \cdot (1 + 2 \cdot 1.005 + 2 \cdot 1.005025 + 1.01005025)}{6} \doteq 1.0100501670 \end{aligned}$$

a stejným způsobem vypočteme následující krok

$$\begin{aligned} k_1 &= f(t_1, y_1) = 1.0100501670 \\ k_2 &= f(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}hk_1) = 1.0100501670 + \frac{0.01 \cdot 1.0100501670}{2} \doteq 1.0151004178 \\ k_3 &= f(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}hk_2) = 1.0100501670 + \frac{0.01 \cdot 1.0151004178}{2} \doteq 1.0151256691 \\ k_4 &= f(t_1 + h, y_1 + hk_3) = 1.0100501670 + 0.01 \cdot 1.0151256691 \doteq 1.0202014237 \\ y_2 &= y_1 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) = 1.0100501670 + \\ &+ \frac{0.01 \cdot (1.0100501670 + 2 \cdot 1.0151004178 + 2 \cdot 1.0151256691 + 1.0202014237)}{6} \doteq \\ &\doteq 1.0202013400 \end{aligned}$$

číslo kroku	t	y
0	0	1
1	0.01	1.0100501670
2	0.02	1.0202013400

Tabulka 2.9: Shrnutí výsledků

2.2.9 Implicitní Eulerova metoda

Explicitní Eulerovu metodu jsme si již představili, jde o velmi jednoduchou, ale také nepřesnou metodu s dost omezenou stabilitou.

$$y_{i+1} = y_i + h \cdot f(t_{i+1}, y_{i+1}) \tag{2.22}$$

Rovnice (2.22) je předpisem implicitní Eulerovy metody. Od explicitní se liší především parametrem funkce $f(t, y)$, kde y_{i+1} je zároveň parametrem funkce i hledanou neznámou. Obecně budeme muset řešit nelineární rovnici (vhodně zvolenou iterační metodou), což je mnohem pracnější než pouhé dosazení do funkce a vyčíslení.

Explicitní i implicitní varianta mají stejnou přesnost, je v něčem tedy implicitní Eulerova metoda výhodnější než explicitní? Hlavním rozdílem je mnohem lepší stabilita metody, viz 2.2.1.

2.2.10 Srovnání numerických metod

Absolutní chyba (popsána v sekci 2.2.1) může být v našem případě zapsána rovnicí (2.23).

$$\varepsilon = |\, y(t_i) - y_i \,| \tag{2.23}$$

V tabulce 2.10 vidíme srovnání velikostí (především řádů) absolutních chyb u jednotlivých metod.

	Eulerova metoda	Heunova metoda	Runge-Kutta 3. řádu	Runge-Kutta 4. řádu [1]
t(0.01)	$5.01671 \cdot 10^{-5}$	$1.67084 \cdot 10^{-7}$	$3.84168 \cdot 10^{-10}$	$8.41681 \cdot 10^{-11}$
t(0.02)	$1.01340 \cdot 10^{-4}$	$3.37527 \cdot 10^{-7}$	$8.26756 \cdot 10^{-10}$	$2.67558 \cdot 10^{-11}$

Tabulka 2.10: Porovnání chyb numerických integračních metod

Z výsledků je dobře patrné, jak jsou jednotlivé metody přesné.

[1] Výsledky RK4 jsou ovlivněny zaokrouhlovacími chybami (příklady ke všem metodám jsem počítal na deset desetinných míst). Při řešení na dvacet desetinných míst je absolutní chyba nižší v prvním kroku o dva řády a v druhém kroku o jeden řád.

Kapitola 3

Návrh řešení

Tato kapitola se zabývá návrhem vlastností aplikace (návrhem jazyka jen okrajově, více se o něm můžete dočíst v [8]). Dále prozkoumáme pár grafických rozhraní simulačních prostředí a shrnu poznatky z jejich používání. V poslední řadě vyložím návrh vzhledu a ovládání editoru blokových schémat.

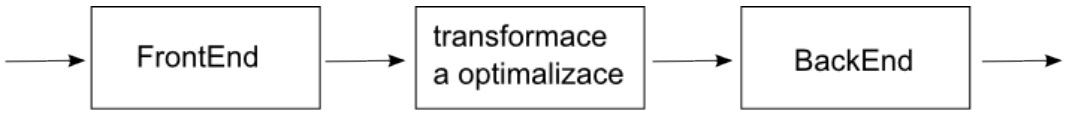
3.1 Návrh gramatiky a vlastností aplikace

Při návrhu gramatiky (a obecně možností programu) jsme vycházeli primárně ze syntaxe jazyka `C` – deklarace proměnných, konstant, definice bloků, if (resp. ternární podmíněný operátor). Některé rysy jsme převzali i ze syntaxe jazyka `C#` (například `out` modifikátor u argumentů bloku, plánovaná syntaxe `foreach` cyklu, polí, sběrnicových argumentů bloku). Zápis diferenciálních rovnic a několik dalších konstrukcí čerpá z TKSL/386. Podrobný popis gramatiky naleznete v příloze (část A).

Při návrhu jsme byli asi příliš optimističtí a přecenili jsme své síly. Část rysů jazyka (hlavně co se týče polí – sběrnic) není implementována, některé další pouze v jednodušší formě (např. místo switch/mux je pouze podmíněný výraz).

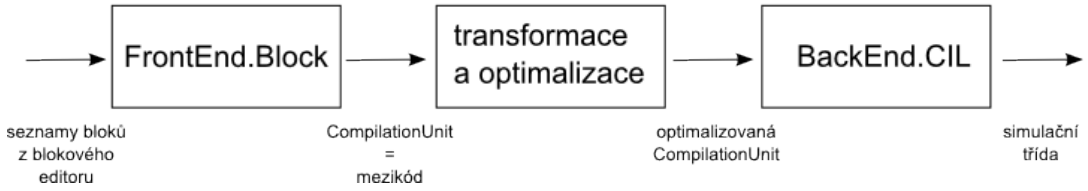
3.2 Postup kompilace

Z gramatiky jazyka jsme postupně vytvářeli návrh mezikódu a dalších tříd, pomocí kterých budou komunikovat velké celky – `FrontEnd` (např. blokový vstup) a `BackEnd` (zatím jediný bude numerický integrátor). Grafické znázornění vidíte na obr. 3.1. V konkrétním



Obrázek 3.1: Postup kompilace souboru

případě blokového vstupu bude postup kompilace vypadat následovně – obr. 3.2. Výstupem `BackEnd.CIL` bude přeložená assembly (knihovna – pod Windows bude mít příponu `dll`) obsahující třídu se simulací.



Obrázek 3.2: Postup kompilace souboru

3.3 Simulování

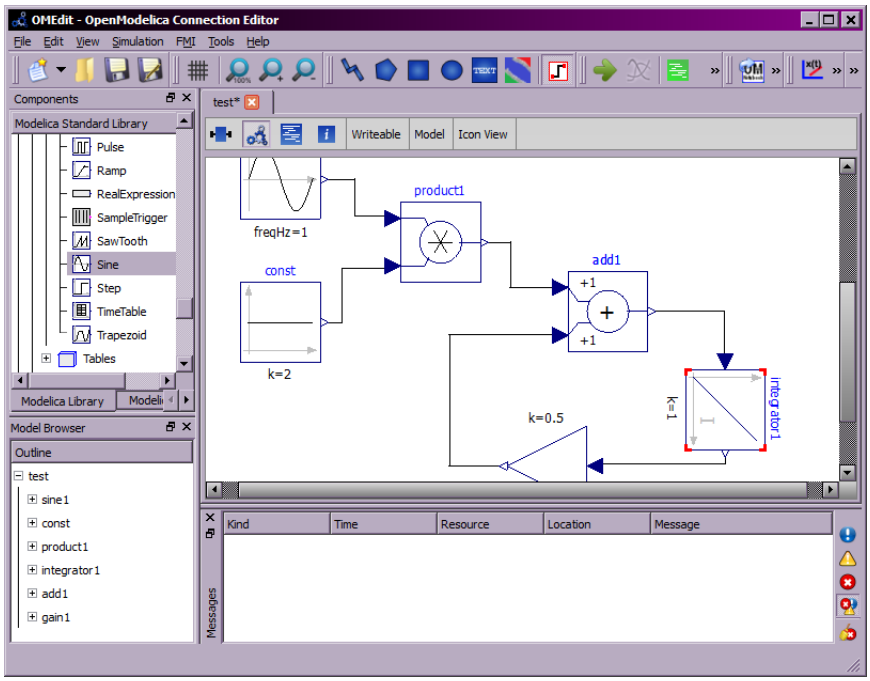
Třída vygenerovaná pomocí `BackEnd.CIL` se instancuje. Nastavíme jí, kam má posílat výsledky z každého kroku, popř. další parametry (krok, metoda) a je připravena k zahájení výpočtů. Zcela jistě bude simulační třída pouštěna v samostatném vláknu, jinak by došlo k nepříjemnému "zatuhnutí"uživatelského rozhraní v průběhu simulace.

3.4 Vzhled bloků, portů a uživatelského prostředí

Nejdříve předvedu jiná simulační prostředí, pak přejdu k návrhu podoby naší aplikace.

3.4.1 OpenModelica

Obr. 3.3 ukazuje rozhraní open source simulačního prostředí OpenModelica. Na levé straně



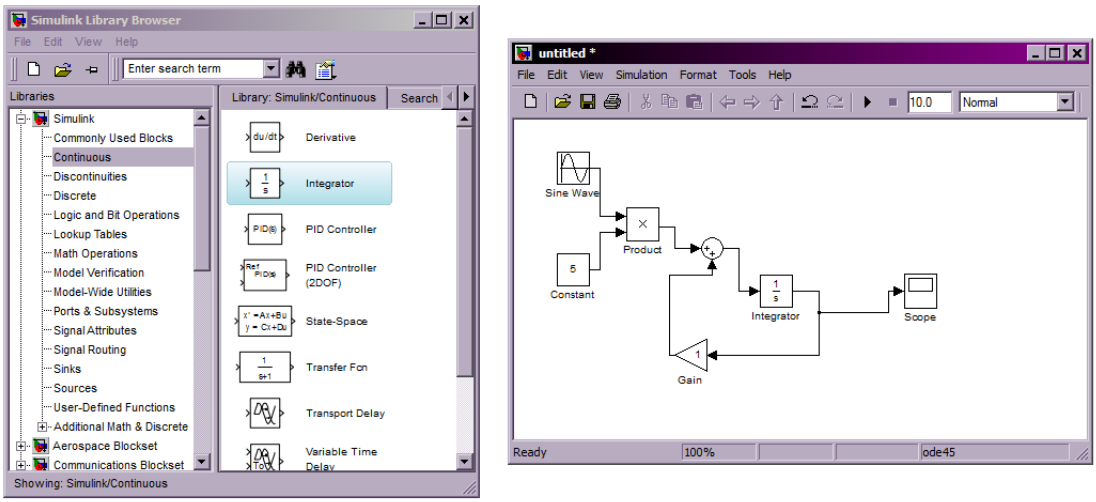
Obrázek 3.3: Uživatelské prostředí aplikace OpenModelica

je seznam všech bloků v podobě stromu, odtud se jednotlivé bloky přetahují na pracovní plochu. Největší část okna zabírá pracovní plocha.

Když jsem zkoušel pracovat s prostředím, tak mi přišlo, že ikonky bloků v seznamu jsou příliš malé (je to asi dáno tím, že Modelica podporuje velké množství různých komponent). Naopak bloky na pracovní ploše na mne působily jako příliš velké.

3.4.2 Simulink

Druhým simulačním prostředím je Simulink, jeho grafické rozhraní vidíme na obrázku 3.4. Zde jedno okno slouží k výběru bloku, je rozdělené na strom kategorií a seznam bloků.



Obrázek 3.4: Uživatelské prostředí aplikace Simulink

Můžeme vidět poměrně velké ikonky bloků v seznamu. Druhé okno je vyhrazeno pro pracovní plochu, bloky se na něj umisťují také přetažením.

Na pracovní ploše jsou bloky poměrně malé, líbí se mi ale jejich jednoduchost a výstižnost.

3.4.3 Návrh vzhledu portů

Postupně jsem zvažoval různé podoby portů, některé z nich jsou k vidění na obrázku 3.5(a). Část z nich je inspirována Simulinkem nebo OpenModelicou.

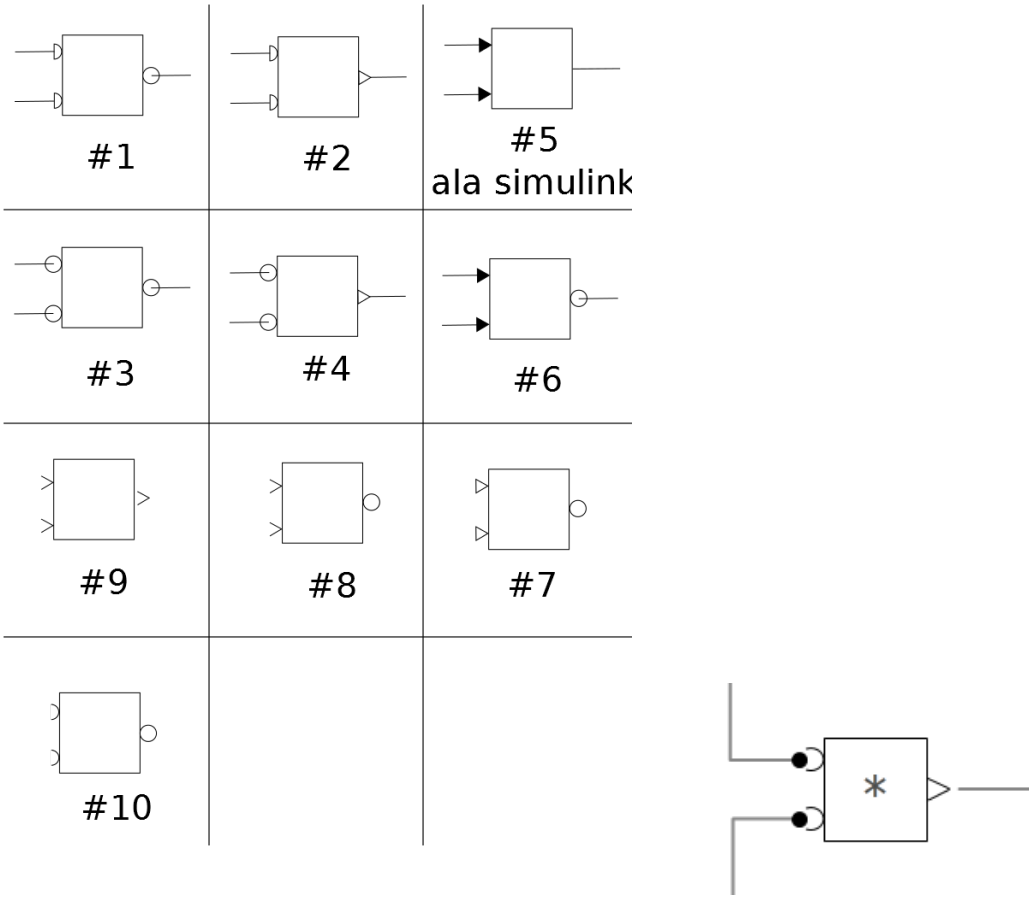
Původně jsem plánoval použít návrh 1. Ten byl ale v průběhu rozhodování vyřazen úplně, protože mít kolečko jako port je matoucí. Celkem často se takto označuje negovaný vstup či výstup.

Nakonec zvítězila kombinace čísla 10 jako vstupu a čísla 9 jako výstupu – výsledek je vidět na obrázku 3.5(b).

3.4.4 Návrh vzhledu bloků

Některé bloky vycházejí ze značení používaného u analogových počítačů (např. integrátor). Snažil jsem se je odlišit i velikostí (bloky s více vstupy, bloky složitější či důležitější bývají větší). U podmíněného bloku jsem zvolil speciální rozmístění portů, abych zdůraznil funkci spodního vstupu – jde o řídící signál (logický výraz podle kterého se rozhodne, který ze vstupů se objeví na výstupu).

Na obr. 3.6 je jeden z návrhů, který je velmi blízký výsledné podobě.



(a) Různé možnosti vstupních a výstupních portů

(b) Finální verze

Obrázek 3.5: Vzhled portů

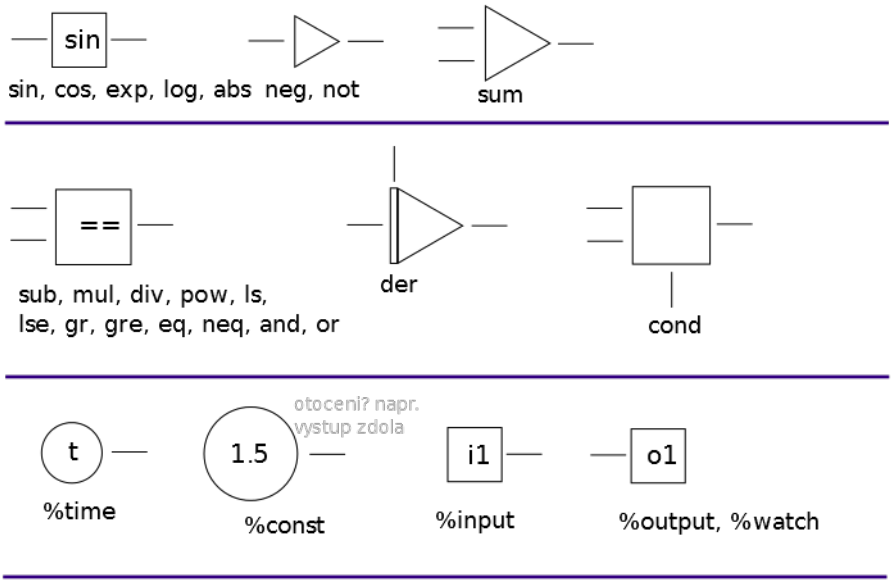
3.4.5 Ovládání a vzhled blokového editoru

Při návrhu rozložení ovládacích prvků v hlavním okně aplikace jsme se inspirovali aplikacemi jako je Visual Studio, Firefox, PSPad. Koncept hlavního okna je na obrázku 3.7.

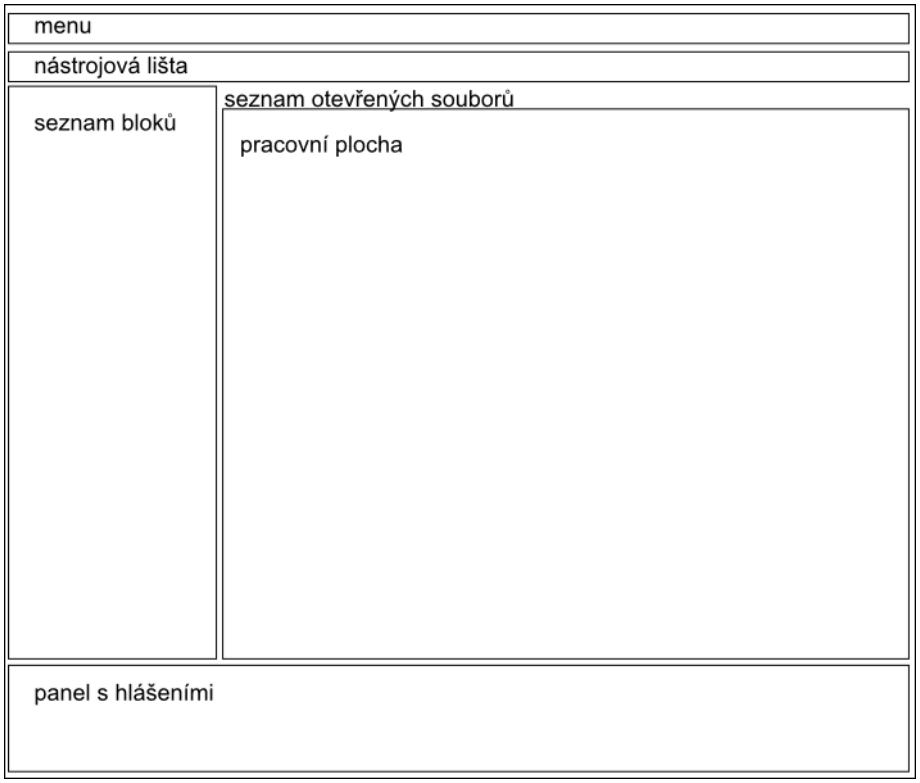
Menu a nástrojová lišta jsou, jak je zvykem, umístěny v horní části. Vlevo je panel s bloky, který bude obsahovat ikonky bloků. Možná bude podporovat seskupování bloků podle kategorie. Seznam otevřených souborů bude seznam záložek ("oušek") se jmény souborů. Po nakliknutí se pracovní plocha přepne na zvolený soubor. Na pracovní ploše bude buďto editor vstupu[1], nebo výsledky (pravděpodobně reprezentované grafem).

Editor bloků se bude ovládat velmi podobně, jako výše představená velká simulační prostředí. Bloky se budou vytvářet přetažením z panelu s bloky (vlevo). Propojení mezi bloky budou řešena opět táhnutím myši, tentokrát ale z výstupního portu do vstupního portu. Blokový editor bude pravděpodobně také provádět kontrolu správnosti propojení (např. nelze spojit výstup s výstupem nebo vytvořit několik spojů do vstupního portu).

[1] Editor vstupu může být textový editor, nebo blokový editor.



Obrázek 3.6: Návrh vzhledu bloků



Obrázek 3.7: Návrh GUI

Kapitola 4

Implementace

Program podporuje dva typy vstupních dat – textová data a tzv. blokové schéma. Textovým vstupem se rozumí zdrojový kód popisující rovnice podobně jako v programu TKSL/386. Rozdíl oproti TKSL/386 je ten, že syntaxe jazyka vychází z rodiny jazyků C a navíc výpočetní systém podporuje rozklad problému na podproblémy – bloky. Výpočetní systém (takže i jazyk textového vstupu) je case-sensitive.

4.1 Bloky

Bloky představují funkční celky obdobně jako v jazyce C. Stejně jako v jazyce C i zde je potřeba uzavřít tělo hlavního výpočtu do speciálního bloku. Tento blok nese označení `system`.

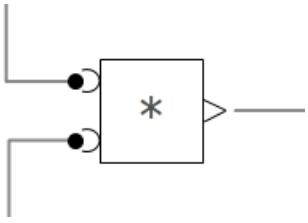
V projektu musí existovat globální konstanty určující parametry simulace.

tmax	čas konce simulace
record	interval záznamu výsledků
step	krok integrační metody
eps	přesnost

Tabulka 4.1: Seznam speciálních globálních konstant

`step` a `eps` se nesmí vyskytnout zároveň. Vždy musí být v projektu definovány právě tři speciální globální konstanty určující parametry simulace.

Detailnější popis uživatelských bloků se nachází v části 4.5.4.

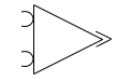
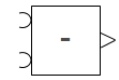
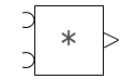
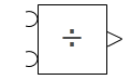
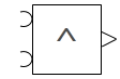


Obrázek 4.1: Ukázka bloku (násobičky) se dvěma zapojenými vstupy a jedním zapojeným výstupem

4.2 Operátory

Drtivá většina operátorů je stejná (významem i značením) jako v jazyce C. Jiný význam má pouze operátor `^`, náš výpočetní systém ho interpretuje jako umocnění (a ne jako bitový XOR).

4.2.1 Aritmetické

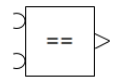
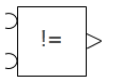
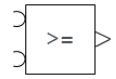
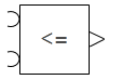
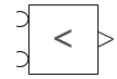
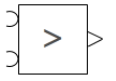
sčítání	`+`	
odčítání	`-`	
násobení	`*`	
dělení	`/`	
umocnění	`^`	

Tabulka 4.2: Podoba aritmetických operátorů

Oproti TKSL/386 je nově podporován i operátor umocnění (`^`), ostatní základní aritmetické operátory fungují stejně jako např. v jazyce `C`.

4.2.2 Relační

Relační operátory vrací hodnotu 1 při úspěchu a hodnotu -1 při neúspěchu.

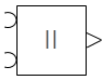
rovnost	`==`		nerovnost	`!=`	
větší, nebo rovno	`>=`		menší, nebo rovno	`<=`	
menší	`<`		větší	`>`	

Tabulka 4.3: Podoba relačních operátorů

Přestože máme operátory rovnosti, není doporučeno je používat.

4.2.3 Logické

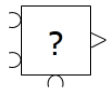
Stejně jako relační i logické operátory vrací 1 pro pravdu a -1 pro nepravdu. Jedinou výjimkou je negace - ta invertuje znaménko (chová se stejně, jako unární mínus).

negace	`!`	
nebo	`or`, `||`	
a zároveň	`and`, `&&`	

Tabulka 4.4: Podoba logických operátorů

4.2.4 Podmíněné

Jde o klasický podmíněný operátor známý z jazyka PHP nebo C, je ale důležité s ním zacházet opatrně. Simulace je totiž implementována velmi jednoduše, neumí například detekovat změnu podmínky a podle toho uzpůsobit krok. Jak bylo zmíněno výše, ostrá rovnost nemusí nikdy nastat (a to ani v případě časové proměnné). Využití podmíněného výrazu je ke generování signálu, který má podmínku jako výraz s časovou proměnnou a ostatní vstupy jsou konstantní (vyčíslitelné při překladu).

podmíněný operátor	`?` `:`	

Tabulka 4.5: Podoba podmíněného operátoru

Podmíněný operátor byl přidán do jazyka proto, aby bylo možné vytvořit např. jednotkový skok.

Na obrázku 4.2 vidíme užití podmíněného operátoru. Výstup bude odpovídat funkci (4.1).

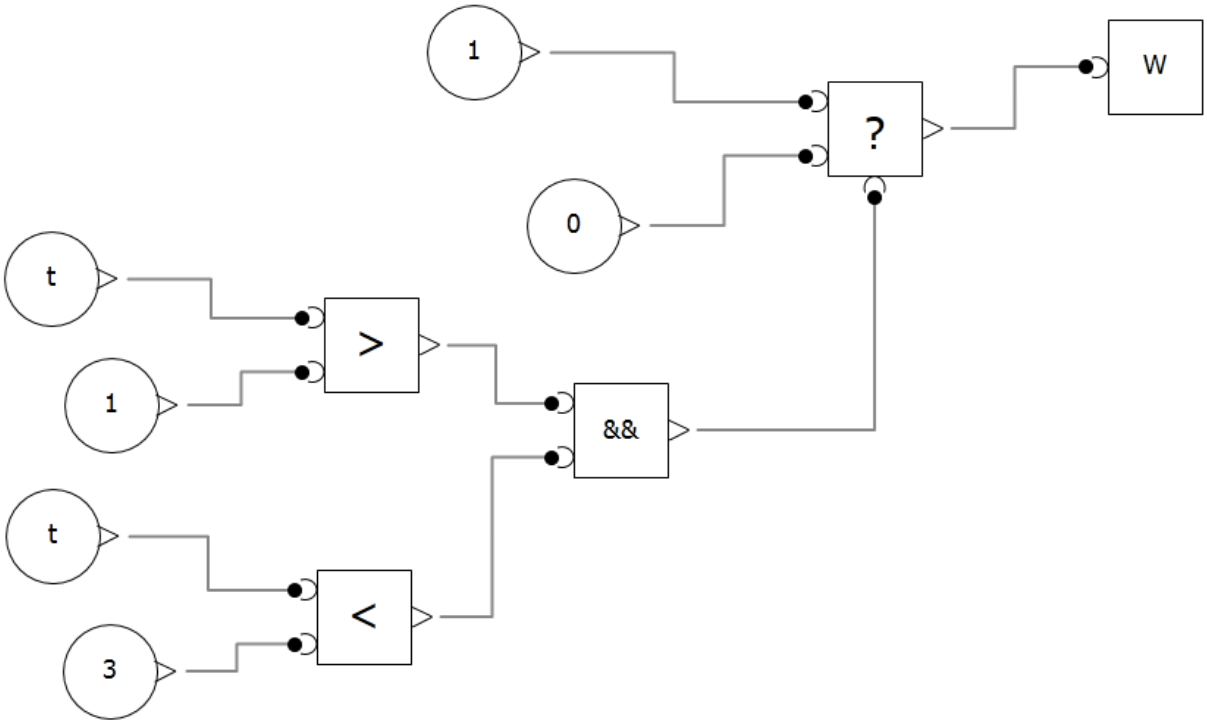
$$f(t) = \begin{cases} 1 & \text{pro } t \in (1,3) \\ 0 & \text{pro } t \notin (1,3) \end{cases} \tag{4.1}$$

4.3 Datové typy

V této části stručně popíšu dva základní datové typy, které se v systému vyskytují.

4.3.1 Proměnná (signál)

Tento datový typ představuje v podstatě vodič, jeho hodnota odpovídá napětí na vodiči. Definovány jsou nad ním operace zmíněné v kapitole 4.2. Kromě přiřazení (v podkapitole 4.5.1) lze užít zápis s derivací proměnné – viz podkapitola 4.5.2.

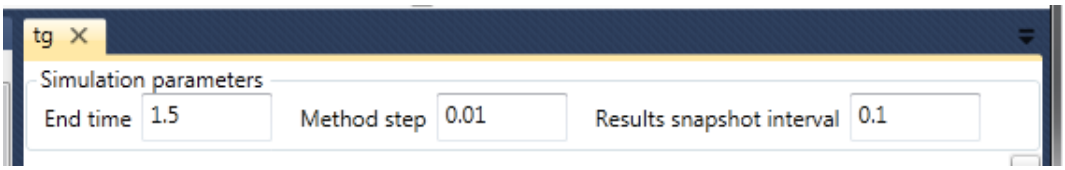


Obrázek 4.2: Schéma příkladu užití podmíněného výrazu

4.3.2 Číselná konstanta

Jde o typ pro anonymní i pojmenovanou konstantu. Vyčísluje se již v době překladu. Rozlišujeme globální a lokální pojmenované konstanty (důsledek zavedení bloků).

Blokový editor podporuje pouze speciální globální konstanty – parametry simulace, viz obr. 4.3.



Obrázek 4.3: Nastavení parametrů simulace v blokovém editoru

4.4 Vestavěné funkce

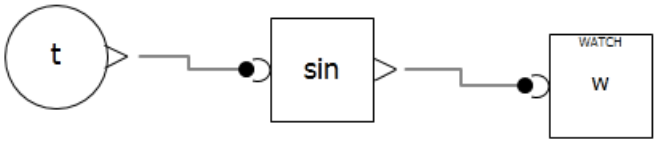
Všechny vestavěné funkce mají jeden vstup a jeden výstup. Používají se stejně, jako jiné bloky (přetáhnou se ze seznamu bloků na pracovní plochu). Seznam všech implementovaných funkcí je v tabulce 4.6, ukázku vestavěné funkce vidíte na obrázku 4.4

4.5 Popis jazykových konstrukcí

Tato sekce pojednává o takových třídách dědících od `ExpressionBase`, které mají povahu příkazu (tj. v textovém zápisu nevracejí hodnotu).

Jméno bloku	Popis
`sin`	sinus
`cos`	kosinus
`ln`	přirozený logaritmus
`exp`	exponenciální funkce o základu e
`abs`	absolutní hodnota čísla

Tabulka 4.6: Seznam vestavěných funkcí

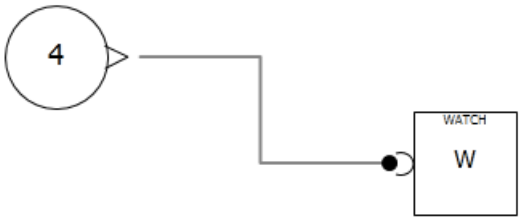


Obrázek 4.4: Schéma s funkcí sinus

4.5.1 Přiřazení

Jde o jednoduché přiřazení – na levé straně je proměnná (reálné číslo) a na pravé straně je výraz. Do každé proměnné může být přiřazeno pouze jednou.

V blokovém editoru nelze přiřazovat konkrétním proměnným (protože neexistují pojmenované vodiče), ale `watch` a výstupnímu bloku přiřadit hodnotu lze (jde jen o propojení lomenou čárou – vodičem).



Obrázek 4.5: Ukázka přiřazení konstanty 4 do `watch` W v podobě blokového schématu

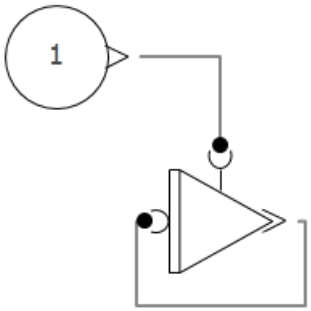
4.5.2 Zápis diferenciální rovnice

Jeden z nejdůležitějších příkazů výpočetního systému, syntaxe vychází z jazyka TKSL/386.

Následující rovnice

$$x' = x \tag{4.2}$$
$$x(0) = 1 \tag{4.3}$$

budou v editoru vypadat takto:



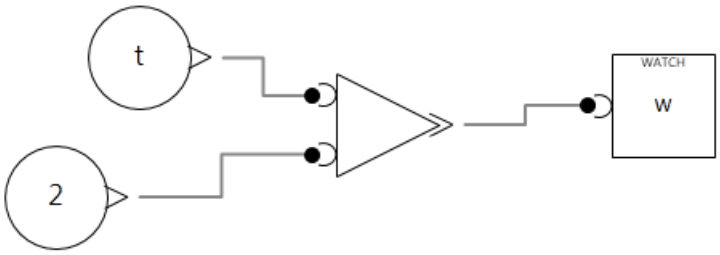
Obrázek 4.6: Ukázka zápisu rovnic (4.2) a (4.3) v podobě blokového schématu

Horním vstupem je počáteční podmínka (v čase $t = 0$), která je vyčíslitelná v době překladu.

4.5.3 Watch

Slouží k záznamu hodnoty výrazu v průběhu simulace. Každá `watch` musí mít přiřazen výraz.

Na rozdíl od jmen konstant, proměnných a uživatelských bloků identifikátory výrazu `watch` musí být unikátní jen vůči ostatním jménům `watch`.



Obrázek 4.7: Ukázka odpovídající funkci $w(t) = t + 2$ v blokovém zápisu

4.5.4 Ostatní příkazy

Část příkazů (`IExpression`) blokový editor neumí vytvářet, ale sémantická kontrola, optimalizace a preprocesory jsou na ně plně připraveny, proto je zde alespoň ve stručnosti zmíním. Více se o nich dočtete v [8].

Cyklus for

Pomocí této konstrukce lze zapisovat určité soustavy rovnic velmi jednoduše. Jako ostatní složitější prvky i tento je implementován formou preprocesoru (tj. postupně se ze stromů s `for` cykly stanou mnohem širší stromy bez `for` cyklů).

Sekvence

Sekvence jsou bloky kódu, nyní je lze použít pouze za `for` cyklem.

Komentáře

Textový vstup podporuje jednořádkové i víceřádkové komentáře ve stylu jazyka `C`.

Definice bloku

Blok se skládá ze jména, parametrů a těla. Vzhledem k funkci jazyka musí mít každý blok alespoň jeden parametr[1]. Blok pouze se vstupním parametrem je voltmetr – v našem jazyce nazvaný jako `watch`, jen s výstupním je generátor signálu.

Parametry bloku jsou dvojího typu - vstupní a výstupní. Vzhledem k tomu, že jazyk pro samotný výpočet nedefinuje více datových typů, plně postačuje u argumentu uvést, zda je vstupní či výstupní.

Volání bloku

Uživatelský blok se bude přidávat do schématu stejně jako bloky vestavěné (např. sčítačka).

4.5.5 Plánovaná rozšíření

Jedná se primárně o příkazy spojené s užitím sběrnic.

Komentáře

V blokovém vstupu by měly mít podobu obdélníků s textem (možná doplněného šipkou).

Sběrnice

Přemýšleli jsme nad typem `counter`, který by sloužil k indexování sběrnic, byl by řídící proměnnou `for` cyklu. Nyní je sice `for` cyklus implementován, ale ve velmi zjednodušené formě – pracuje na principu přeznačování proměnných podle aktuální iterace.

$$\begin{aligned} x' &= 1 + x \\ x(0) &= 1 \end{aligned} \tag{4.4}$$

S tělem cyklu předepsaným rovnicemi (4.4) a parametry cyklu, kde určíme x jako vnitřní proměnnou a zadáme 2 iterace, dostaneme soustavu (4.5).

$$\begin{aligned} x1' &= 1 + x1 \\ x1(0) &= 1 \\ x2' &= 1 + x2 \\ x2(0) &= 1 \end{aligned} \tag{4.5}$$

Tímto způsobem lze zapsat jen velmi omezené množství reálných problémů. Proto se uvažovalo nad `for` cyklem, který je zápisem podobný jazyku `C`. Samozřejmě potom by bylo

[1] Navíc u uživatelem definovaných bloků mají smysl jen ty, které jsou výstupní - mají alespoň jeden výstupní parametr (uvnitř bloku nelze užít watch, takže cokoliv by se spočítalo uvnitř, ven z volání bloku by neprobublalo.)

potřeba přidat i nějakou formu `if` příkazu, který by stejně jako `for` byl vyhodnocován v době překladu a umožnil by tak zadávat velké soustavy mnohem pohodlněji.

S tímto samozřejmě souvisí podpora předávání proměnných typu sběrnice blokům. S přidáním sběrnic by bylo možné zavést vícevstupé základní bloky (např. sčítačku, logické nebo atd.), což by značně zjednodušilo a zpřehlednilo práci v editoru blokových schémat.

Uživatelské bloky

Datová vrstva i blokový parser má plně funkční implementaci uživatelem definovaných bloků, bohužel blokový editor (GUI) podporuje zatím pouze systémový blok.

Switch

V současném stavu máme implementován podmíněný výraz (podkapitola 4.2.4), který ale není tak mocný, jako plánovaný `switch`. Opět vychází z jazyka `C`. V našem podání by měl ale umět i intervaly (včetně těch, které mají jednu mez nekonečno).

Více operátorů a vestavěných funkcí

V hodně raných verzích aplikace byl k dispozici operátor modulo, v jiných jazycích často značený jako `%`. Se zavedením sběrnic by bylo nutné ho implementovat (pro indexování polí). Myslím si ale, že i kdyby nedošlo na zavedení polí, mohl by i tak být zajímavý pro (snadnější[2]) generovaní periodických signálů.

Z vestavěných funkcí, které by stály za zvážení, zda je nepřidat, bych zmínil např. tangens a kotangens[3]. Možná i funkce signum by našla uplatnění (nyní kombinací podmíněných bloků půjde sestavit, ale nativní řešení by bylo vhodnější).

4.6 Vnitřní reprezentace dat

V této části bych rád vyjmenoval a stručně popsal nejpodstatnější třídy, které jsou nezávislé na back-endu a front-endu.

4.6.1 `Expression`

Základní stavební blok stromu výrazů a příkazů. Každá třída dědící od `ExpressionBase` (a tím pádem i od `IExpression`) umí vrátit počet svých podvýrazů, svoje podvýrazy (zase jde o `IExpression`), svůj typ (např. `Constant`), hodnotu (užívá se třeba u konstant, odkazů na proměnné), pozici v původním zdroji (u textu řádky a sloupce, u blokového grafického vstupu jde o čísla bloků/drátů v `BlocksManager`) a je schopna vytvořit svoji kopii.

4.6.2 `CompilationUnit`

Výstupem parseru (ať už textového nebo blokového) je `CompilationUnit`. Vnitřní struktura objektu je velmi podobná textovému vstupu. Jsou zde seznamy globálních konstant, bloků a system bloku. Každý blok (i ten hlavní - system blok) má svoje proměnné, konstanty, watch a příkazy (v jednoduchých případech jsou to pouze rovnice).

[2] Program umí sinus a podmínku, takže nějaké periodické signály určitě půjdou poskládat.

[3] Nyní sice jdou vypočítat, ale pouze za užití funkcí sinus a kosinus – $tg(x) = \frac{sin(x)}{cos(x)}$ a $cotg(x) = \frac{cos(x)}{sin(x)}$.

4.6.3 `Results`

Aktuálně implementovaný třídou `ListOfArraysResults`, data zde jsou v listu (seznamu) polí. Pole jsou zde užity záměrně kvůli výkonu (použití kolekce by bylo podstatně náročnější). Do výsledků se ukládají stavy všech proměnných v diferenciálních rovnicích, stavy všech pomocných proměnných (využívají se u vestavěných funkcí jako je např. sinus.) a hodnoty všech `watch`.

4.6.4 `Project`

Drží údaje o všech souborech projektu a jejich typech. Více se o této třídě dočtete v [8].

4.7 Textový vstup

Zde představím třídy zodpovědné za zpracování vstupu, který je v textové formě.

4.7.1 Scanner

Třída `BasicScanner` je běžný konečný automat (s pár pomocnými metodami). Na vstupu (při inicializaci) očekává `TextReader`. Výstupem je token, který parser získává voláním metody `NextToken`. Dále scanner obsahuje čítač řádků, čítač sloupců a příznak čtení komentářů (metadat).

Detailnější popis naleznete v [8].

4.7.2 Parser

Vyžaduje na vstupu `IParserContext`, který může nést projektovou `CompilationUnit` a jméno zpracovávaného souboru (používá se pro chybová hlášení), musí obsahovat inicializovaný `IScanner`.

Více o těchto třídách (včetně příkladů) lze najít v [8]. Pokud se chcete dozvědět podrobnější informace o různých možnostech implementace parseru (a o teoretickém pozadí), tak zkuste např. [11].

Hlavní parser

Je implementován metodou syntaktické analýzy shora dolů (použit rekurzivní sestup). Gramatika je typu LL(1) – bere vstup zleva a vytváří nejlevější derivaci. Hlavní parser se stará v podstatě o vše kromě výrazů (v užším slova smyslu, nejde o `IExpression`).

Expression parser

Když hlavní parser očekává na vstupu výraz, zavolá expression parser. Tento parser užívá metodu analýzy zdola nahoru – od koncových uzlů (terminálů) postupným aplikováním redukcí dojdeme k počátečnímu symbolu gramatiky. K rozhodování, kdy má dojít k redukci je použita precedenční tabulka.

4.8 Blokový vstup

Nyní se budu věnovat blokovému vstupu, který uživatel zadává pomocí blokového editoru[4].

4.8.1 `BlockStore`

Jde o třídu nesoucí všechna data o blocích, vodičích, konstantách. `BlockStore` uchovává vnitřní strukturu právě jednoho bloku (system bloku, nebo uživatelem vytvořeného bloku). Třída sama o sobě nic neumí, slouží pouze jako kontejner pro kolekce dat.

4.8.2 `BlocksManager`

O všechny operace nad datovou vrstvou bloků se stará `BlocksManager`.

Využití

- vytvoření bloku – při zadání pouze jména, automaticky prohledá všechny bloky (i uživatelské) a vytvoří blok s odpovídajícím jménem a příslušným počtem portů
- propojení dvou portů bloků – zde se kontroluje např. zda už vstup není obsazen
- zrušení propojení – odpojení jednoho vodiče, nebo všech od zadaného bloku
- uložení do souboru, načtení ze souboru
- nastavení parametrů bloku – např. jméno `watch`, hodnota anonymní konstanty atd.
- detekce změny předpisu uživatelem definovaných bloků
- kontroly – volané `BlockParser`em před začátkem kompilace do mezikódu (např. jestli jsou obsazeny všechny porty, zda nejsou duplicitní jména `watch`, jestli jsou vyplněny všechny potřebné parametry u bloků)

4.8.3 `BlockParser`

Jeho úkolem je převést blokovou reprezentaci na rovnicovou (která je skoro totožná s textovým vstupem).

Nejdříve se vykoná několik kontrol (ta nejpodstatnější je kontrola rychlých smyček), pak se všem výstupům bloků vygenerují označení (později se z nich stanou pomocné proměnné). Nyní se vygenerují volání bloků (např. pro sčítačku se vygeneruje volání triviálního bloku jménem "sum"s parametry odpovídajícími dříve vygenerovaným pomocným jménům proměnných) tak, že každé volání musí mít známé všechny vstupní parametry (kromě integrátorů, mají vnitřní paměť a proto "oddělují" vstupní a výstupní porty). Na konci proběhne ještě několik jednodušších úkonů – doplnění informací o vstupech a výstupech pokud jde o uživatelský blok, vytvoření informací o globálních i lokálních konstantách, v případě system bloku se vytvoří seznam `watch` a pro oba typy bloků se přidají informace o pomocných proměnných.

Tímto `BlockParser` dokončil generování `CompilationUnit`, která je nyní připravena pro kontroly, transformace, optimalizace a pro následné předání back-endu (viz. 4.14.2).

[4]Lze i ručně tvořit soubory obsahující schémata (formát XML), které následně může program otevřít. Není to ale doporučovaný postup, proces tvoření je velmi zdlouhavý a nepřehledný.

4.9 Sémantická kontrola

Třída `SemanticChecker` postupně volá jednotlivé kontroly.

4.9.1 `FlagsSetter`

Zde probíhá kontrola základních informací v `CompilationUnit` (dále jen CU). Nejdříve se zahodí všechny informace o použití konstant a proměnných (např. zda mají hodnotu, jestli byly čteny, atp.). Pak se prochází všechny seznamy stromů příkazů (`IExpression`) – tj. všechny uživatelské bloky, system blok a všechny `watch` v system bloku. Všechny výrazy a podvýrazy se procházejí a značí, zda byly čteny, zda mají definovanou hodnotu, kontroluje se existence identifikátorů (globálních i lokálních) konstant, proměnných a bloků. Dále se provádí kontrola parametrů volání bloků (počet a typy musí odpovídat předpisu), značí se built-in bloky.

4.9.2 `FlagBasedChecker`

Slouží hlavně ke generování varování o nepoužitých proměnných, konstantách a uživatelem definovaných bloků.

4.9.3 `WalkerChecker`

Představuje druhý průchod stromy příkazů. Nyní už máme dostatek informací o tom, které proměnné budou mít definovanou hodnotu (v prvním průchodu by byl problém např. se soustavami diferenciálních rovnic – výstup druhého integrátoru by nemohl být použit jako vstup prvního, protože druhá proměnná ještě není označena příznakem existence hodnoty/přiřazení).

Příklad

$$x' = y \tag{4.6}$$
$$x(0) = 0 \tag{4.7}$$
$$y' = -x \tag{4.8}$$
$$y(0) = 1 \tag{4.9}$$

Při prvním průchodu (parserem) nelze u pravé strany rovnice (4.6) říci, jestli proměnná y je zapsaná v derivované formě, nebo uživatel zapomněl na naplnění hodnotou. Pokud je označena jako derivovaná, pak je vše v pořádku (to ale lze zjistit až při druhém průchodu). Pokud ale jde o normální proměnnou, tak program (především kvůli přehlednosti) vyžaduje, aby proměnná byla nejdříve definována a až potom použita.

Tyto testy se týkají především textového vstupu, u blokového vstupu toto zajišťuje samotný parser.

4.9.4 `WatchesChecker`

Primárně kontroluje, jestli "nonsimple"`watch`[5] nemá stejné jméno jako proměnná (generuje se jen varování).

[5] "Non-simple"`watch` je taková `watch`, která není definována pouze jediným `Expression` typu čtení hodnoty proměnné (identifikátor watch a proměnné jsou stejné).

4.10 Simulace

Tato část se zabývá přípravou, vznikem a fungováním simulační třídy.

4.10.1 Transformace vstupních dat

Vstupem je zkompilovaný kód z parseru v podobě instance třídy `CompilationUnit`, která je popsána v podkapitole 4.6.2.

Postup transformace

1. kontrola integrity – podrobněji v části 4.11.3
2. nastavení příznaků (kromě `for` cyklů) – viz 4.9.1
3. zpracování volání trivial bloků – viz část 4.11.6
4. rozgenerování `for` cyklů – podrobněji v části 4.11.4
5. odstranění sekvencí vzniklých rozgenerováním `for` cyklů – viz 4.11.5
6. první sémantická kontrola – viz 4.9
7. výpočet globálních a následně i lokálních konstant – má na starosti třída `RES`, viz. podkapitola 4.11.1
8. rozgenerování bloků – implementováno třídou `BlockPreprocessor` 4.11.2
9. pro system blok proběhnou závěrečné úpravy (o všechny tyto modifikace se stará `RES` – podrobnější informace v části 4.11.1)
10. závěrečná sémantická kontrola

4.10.2 Vznik simulační třídy

Při úspěšném převedení `CompilationUnit` do optimalizované podoby se vytvoří kód třídy dědící od `SimulationBase`, který se následně přeloží. Díky tomu jsou výpočty prováděné při řešení velmi rychlé (jde o kód, který je velmi blízký nativnímu).

4.10.3 Samotná simulace

Po nastavení simulace (nad třídou dědící od `SimulationBase`) se zavolá metoda `Run`. Pomocí událostí se výsledky delegují vyšší vrstvě (odkud jsou předávány třídě implementující `IResults`). Podrobněji popsané vrstvy nad simulací můžete nalézt v [8].

Simulační třída je implementována velmi jednoduše. Podporuje pouze prodloužení posledního kroku, pokud je příliš malý (aby se předešlo zaokrouhlovacím chybám).

Nyní program umí následující numerické metody: Eulerovu, Heunovu, Runge-Kutta 3. a 4. řádu. Jako vše ostatní i simulační třída je modulární, snadno lze přidat další (jednokrokové) integrační metody.

Výsledky (`watch`) se vypočítávají po dokončení simulace (většina tříd je už připravena na souběžný výpočet se simulací).

Při implementaci třídy `SimulationBase` a numerických metod jsem čerpal primárně z [13].

4.11 Pomocné třídy

Několik dost podstatných tříd jsem ještě neuvedl. Jde především o třídy provádějící transformace `CompilationUnit` na tvar vyžadovaný pro úspěšné vytvoření simulační třídy.

4.11.1 `RecursiveExpressionsSimplifier`

Hlavní funkcí této třídy je transformace `CU` do takové podoby, ze které může `BackEnd` (viz. podkapitola 4.14.2) snadno vygenerovat požadovaný výstup.

Popis chování metody `OptimizeCompilationUnit`[6]:

1. dosazení konstant do stromů výrazů
2. zjednodušení (částečné vyčíslení)
3. dosazení nederivovaných proměnných do diferenciálních rovnic
4. odstranění nederivovaných proměnných
5. zjednodušení výrazů (předpočítání konstantních výrazů)
6. předpočítání výrazů ve `watch`

Dále řeší ještě jednu úlohu (podstatně jednodušší) – vyčíslení konstant (a následné dosazení vypočtené hodnoty do všech výskytů čtení této konstanty).

4.11.2 `BlockPreprocessor`

Slouží k rozgenerování uživatelských bloků. Postupně se provedou tyto úkony:

1. seřazení bloků za sebe tak, aby se žádný předcházející neodkazoval na následující (rekurze se nesmí vyskytnout)
2. postupně pro každý blok se vykonají následující úkony
   (a) všechna volání bloků se rozgenerují (tj. zkopíruje se tělo bloku)
   (b) nahradí se všechny identifikátory, které se vyskytují v rozgenerovávaném bloku (novými unikátními identifikátory)
   (c) nahradí se všechny parametry bloku těmi výrazy (nebo proměnnými), které byly užity při volání bloku

4.11.3 `IntegrityChecker`

Jeho funkcí je kontrola výstupu `FrontEndu` (obecného, ne jen text a block) – tzn. kontrola `CompilationUnit`.

Provádí ověření důležitých konstant – správnost parametrů simulace, správnost jmen parametrů simulace [7] . Dále se nad všemi uživatelskými bloky a system blokem zkontroluje existence vnořených tříd, pak následuje kontrola všech výrazů ve všech stromech bloku.

[6] V této fázi transformace `CU` už neexistují žádné uživatelské bloky, všechny byly "rozgenerovány"přímo do `system` bloku.

[7] První `FrontEnd` užitý na `CompilationUnit` nastaví jména parametrů simulace – nyní jsou to `eps, tmax, step, record`.

Primárně se kontroluje shoda typu instance a vlastnosti typ v instanci výrazu, úplnost stromu (nelze mít např. sčítačku jen s jedním vstupem) a vlastnost hodnota výrazu (např. u přiřazení nese jméno proměnné) – v některých typech výrazů je vyžadován neprázdný řetězec, jinde nesmí tato vlastnost vracet nic jiného než `null`.

4.11.4 `ForPreprocessor`

Na vstupu dostane `CompilationUnit` (stejně jako ostatní preprocesory). Projde seznamy příkazů ve všech blocích a každý `for` příkaz nahradí příkazem `sequence`, který v sobě obsahuje seznamy příkazů odpovídající jednotlivým iteracím cyklu.

4.11.5 `SequencePreprocessor`

Jeho úkolem je odstranit příkazy `sequence` tak, že vloží těla sekvencí přímo do seznamu příkazů.

Příklad

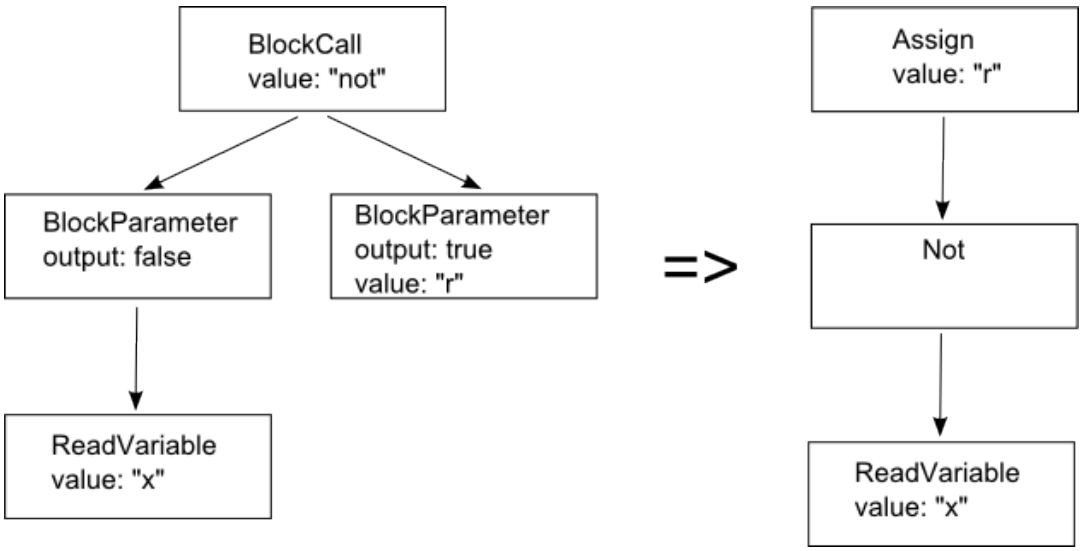
Mějme seznam příkazů `{ příkaz #1, sekvence #2, příkaz #3 }`, kde sekvence obsahuje `{ příkaz #4, příkaz #5 }`. Po odstranění sekvencí bude seznam příkazů vypadat takto: `{ příkaz #1, příkaz #4, příkaz #5, příkaz #3 }`.

4.11.6 `TrivialBlockPreprocessor`

V podstatě jde o převedení zápisu základních operací z formy volání bloku do formy příkazu.

Příklad

Na obr. 4.8 vidíme příklad funkce `TBP` při zpracovávání volání bloku negace.



Obrázek 4.8: Příklad funkce `TrivialBlockPreprocessoru`

4.12 Výsledky simulace

Podle uživatelem zvoleného typu výstupu se vygeneruje požadovaný soubor s výsledky, nebo v případě grafického rozhraní se zobrazí v příslušné záložce.

Seznam typů výstupu

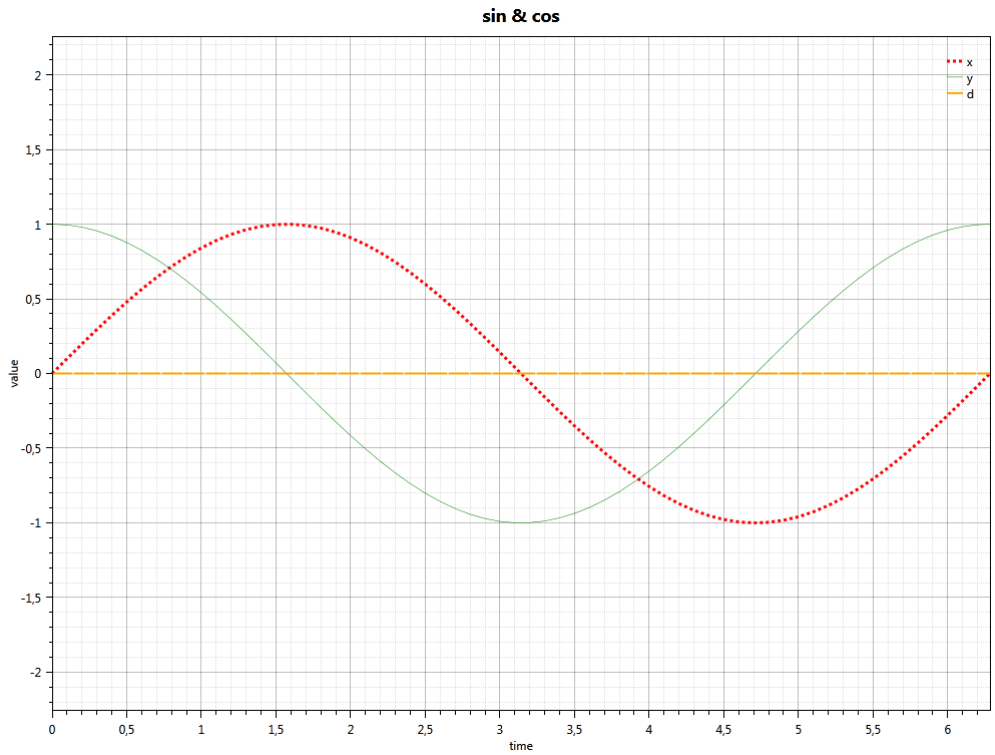
1. Textové
   CSV
2. Grafické
   PNG, BMP, JPEG, TIFF, GIF, SVG

Příklady výstupu

Příklad neúplného textového CSV výstupu:

```
t,0.0E00,1.0E-01,2.0E-01,...
x,0.0E00,9.98334332718592E-02,1.98669363795956E-01,...
```

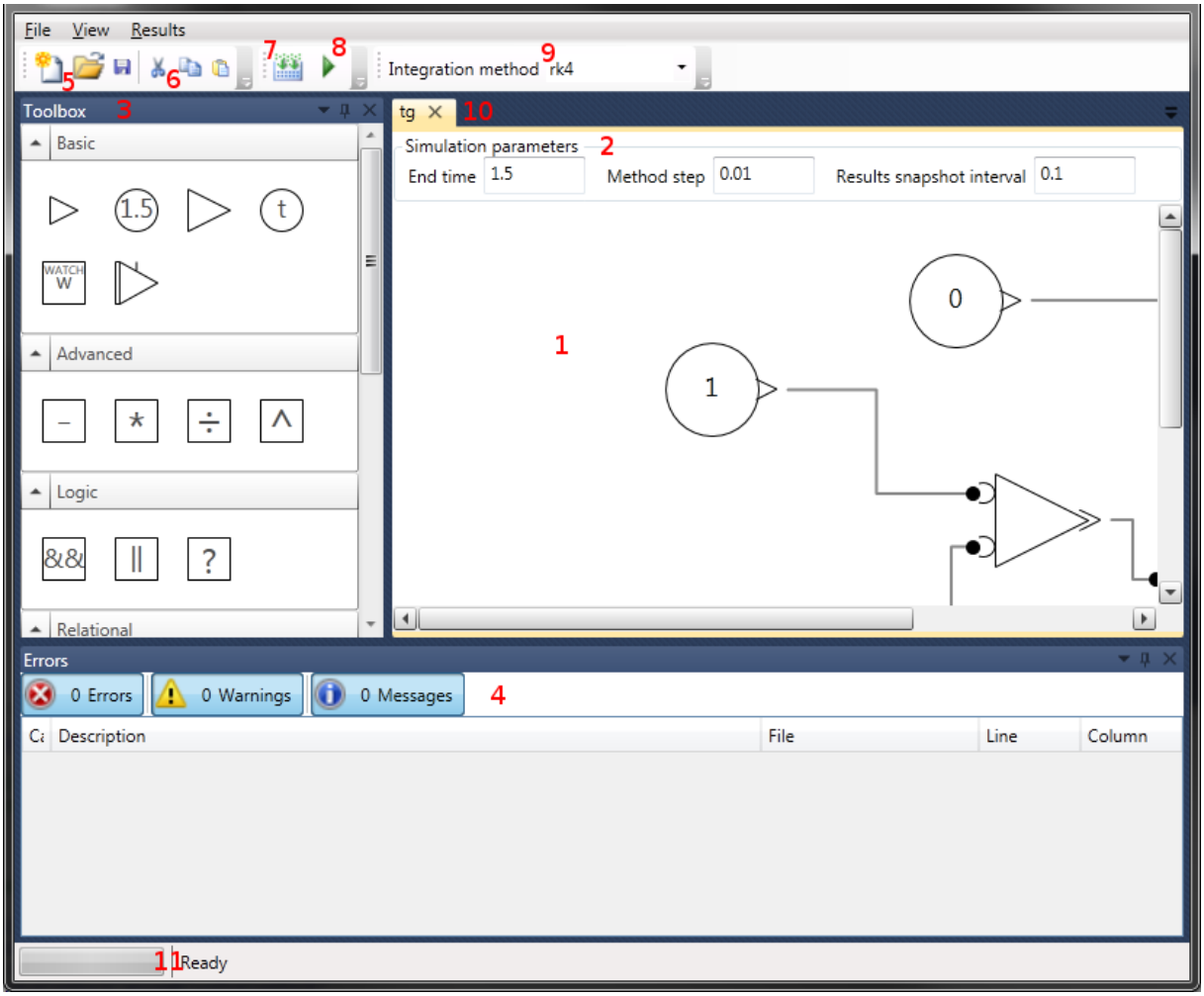
Na obr. 4.9 vidíme příklad grafického výstupu.



Obrázek 4.9: Ukázka vyexportovaného grafu z aplikace

4.13 Grafické uživatelské rozhraní blokového vstupu

Na obrázku 4.10 si můžeme prohlédnout grafické uživatelské rozhraní, konkrétně editor blokových schémat.



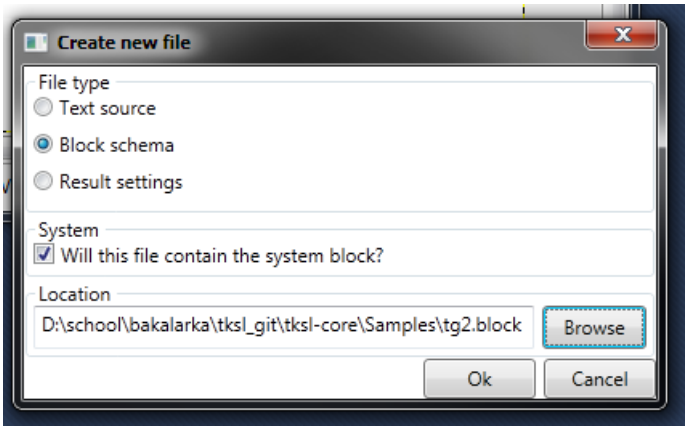
Obrázek 4.10: Grafické uživatelské rozhraní (GUI)

Popis jednotlivých částí rozhraní:

1. plocha – na plochu se přetahují komponenty z panelu s bloky, lze vytvářet propojení a u určitých bloků nastavovat jejich vlastnosti
2. nastavení simulace – koncový čas, krok metody a krok záznamu výsledků
3. panel s bloky – zde jsou všechny dostupné bloky, na plochu se přidávají přetažením
4. chybová hlášení – zobrazují se tu např. chyby a varování při překladu
5. operace se soubory – tlačítka pro nový soubor, otevřít a uložit
6. schránka – vyjmutí, kopírování a vložení
7. kompilovat
8. spustit simulaci
9. volba integrační metody

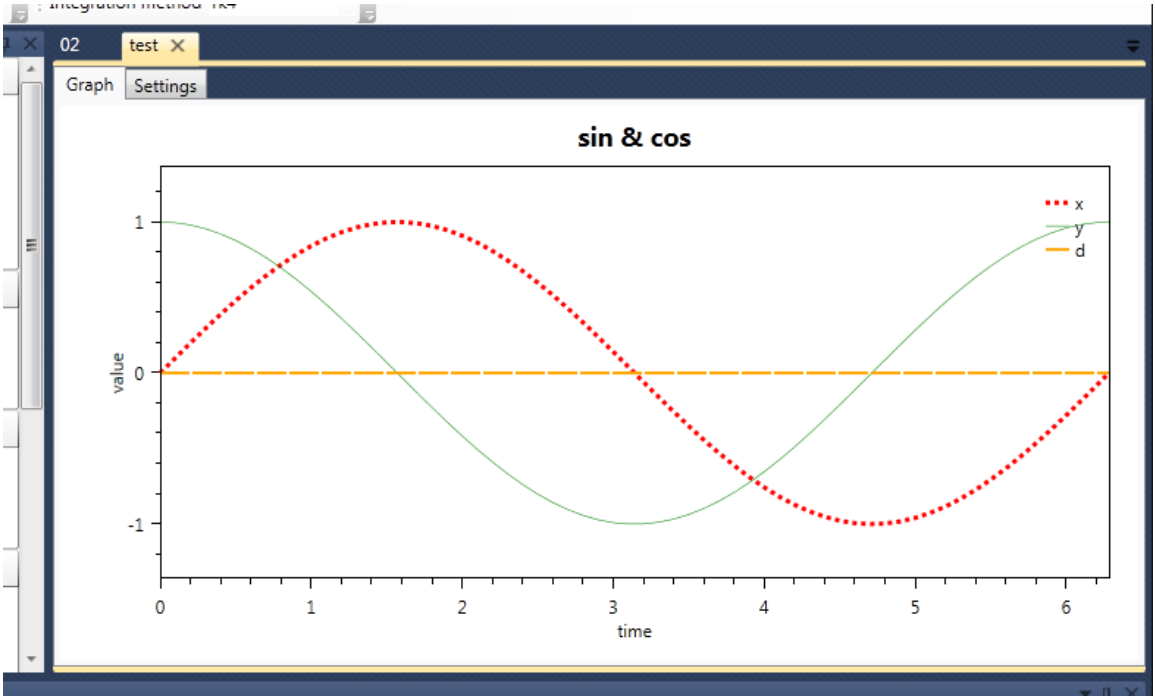
10. seznam otevřených souborů – jde o klasické "taby", lze přepínat mezi jednotlivými soubory, kliknutím na křížek se soubor zavře

11. stavová lišta – skládá se z progress baru (zobrazuje např. stav simulace) a informačního textu



Obrázek 4.11: Dialog vytvoření nového souboru

Obrázek 4.11 ukazuje vytvoření nového souboru, na obrázku 4.12 vidíme graf s výsledky simulace.



Obrázek 4.12: Záložka s výsledky

4.14 Zastřešující třídy

O všech vrstvách a jejich komunikaci se více dozvíte v [8].

4.14.1 Front-end

Jde o zdroj dat, výstupem je `CompilationUnit`. Nyní máme dva frontendy, jeden pro textový vstup (`FrontEnd.Text`) a druhý pro blokový vstup (`FrontEnd.Block`).

4.14.2 Back-end

Úkolem back-endu je přeložit optimalizovanou `CompilationUnit` do výsledného kódu. Aktuálně je implementován pouze jeden backend a tím je `BackEnd.CIL`. Jeho druhá část (`BackEnd.CIL.Runtime`) se stará o samotnou simulaci – jsou zde např. třídy `SimulationBase` a `EulerIntegrationMethod`.

4.14.3 `Compiler`

Na vstupu očekává front-end, back-end a nastavení. Získá `CompilationUnit` z front-endu, provede nutné transformace a optimalizace (viz 4.10.1). Nakonec vše předá back-endu a výsledek vrátí.

4.14.4 `Simulator`

Vstupem této třídy je nastavení simulace a simulace samotná (třída dědící od `SimulationBase`). Zastřešuje práci s výsledky – v průběhu simulace je plní, po skončení simulace zavolá výpočet `watch`. Tato třída se také stará o posílaní zpráv o simulaci vyšším vrstvám, např. stav simulace (v procentech) nebo úspěšné dokončení simulace.

Kapitola 5

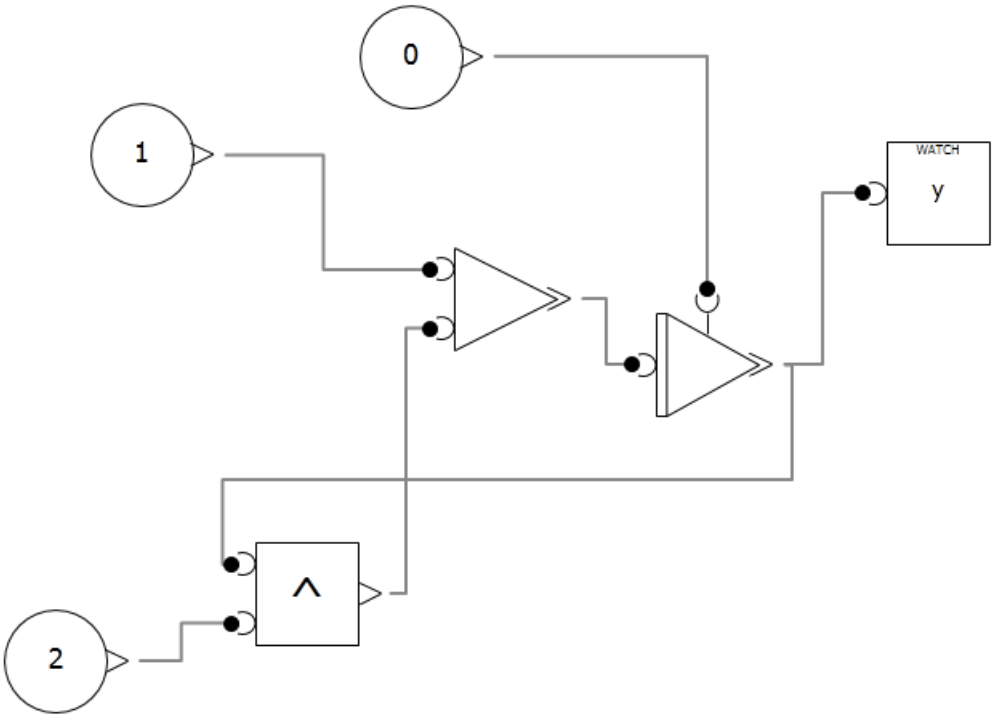
Zhodnocení dosažených výsledků

V této kapitole nejdříve provedu srovnání přesnosti implementovaných metod ve výsledné aplikaci, pak předvedu funkčnost na několika jednoduchých příkladech (čerpáno z [16]).

5.1 Srovnání přesnosti s vestavěnou metodou v .NET

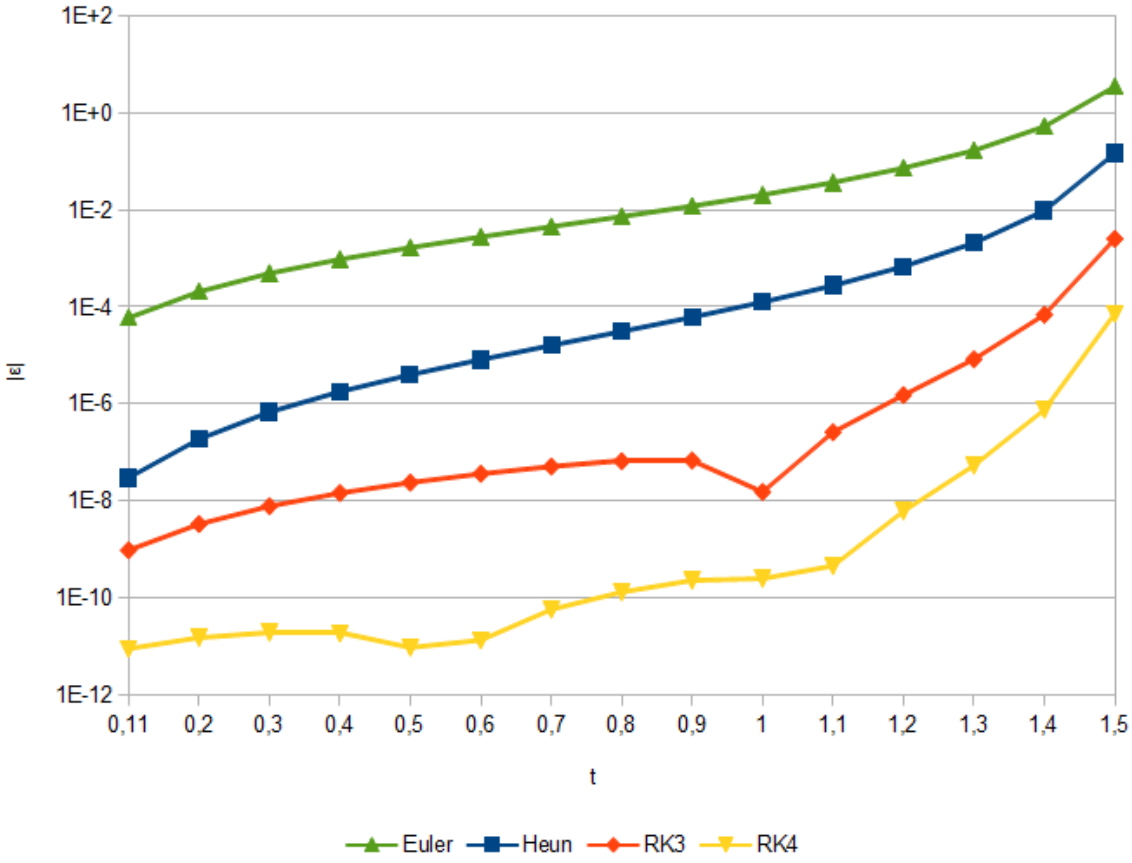
Pro simulaci byl zvolen krok $h = 0.01$. Pro srovnání bude sloužit funkce tangens, její předpis diferenciální rovnicí vidíme na (5.1) a ve formě blokového schématu je na obrázku 5.1.

$$\begin{aligned} y' &= 1 + y^2 \\ y(0) &= 0 \end{aligned} \tag{5.1}$$



Obrázek 5.1: Blokové schéma odpovídající soustavě (5.1)

Následuje graf 5.2 zobrazující rozdíl mezi naším řešením a řešením pomocí zabudované metody ve frameworku (pro chybu je použito logaritmické měřítko).

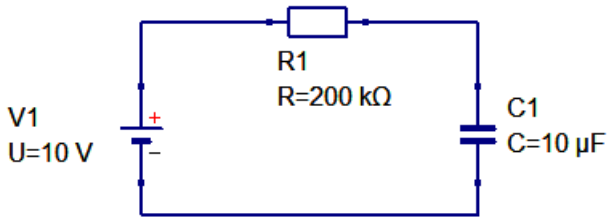


Obrázek 5.2: Graf absolutní hodnoty chyby implementovaných metod

5.2 Vybrané příklady

Nyní předvedu několik jednoduchých příkladů a představím srovnání s jinými simulačními systémy.

5.2.1 RC obvod – nabíjení



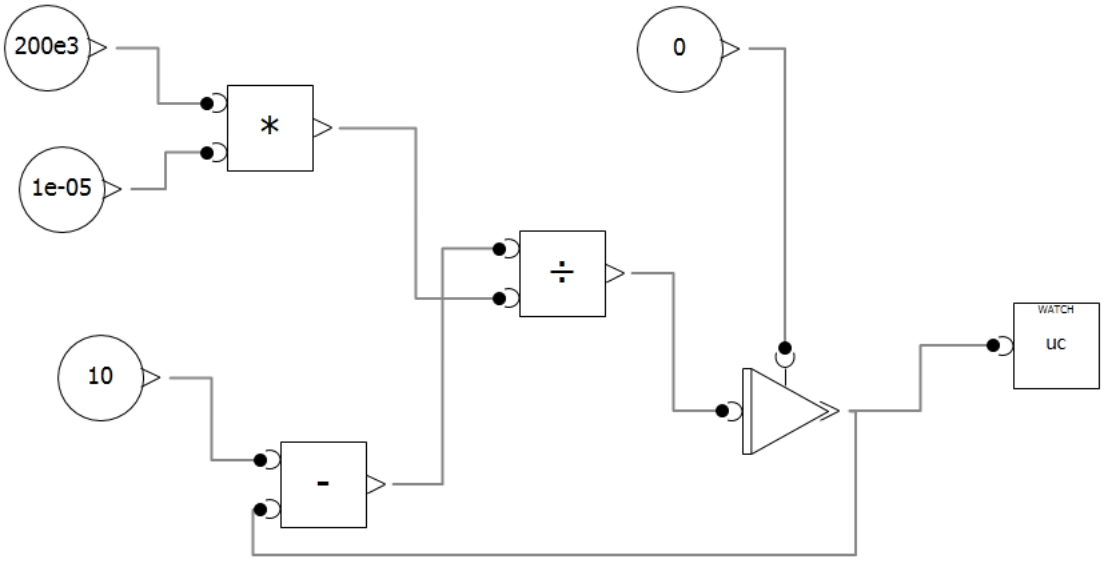
Obrázek 5.3: Obvod RC

Mějme zadání

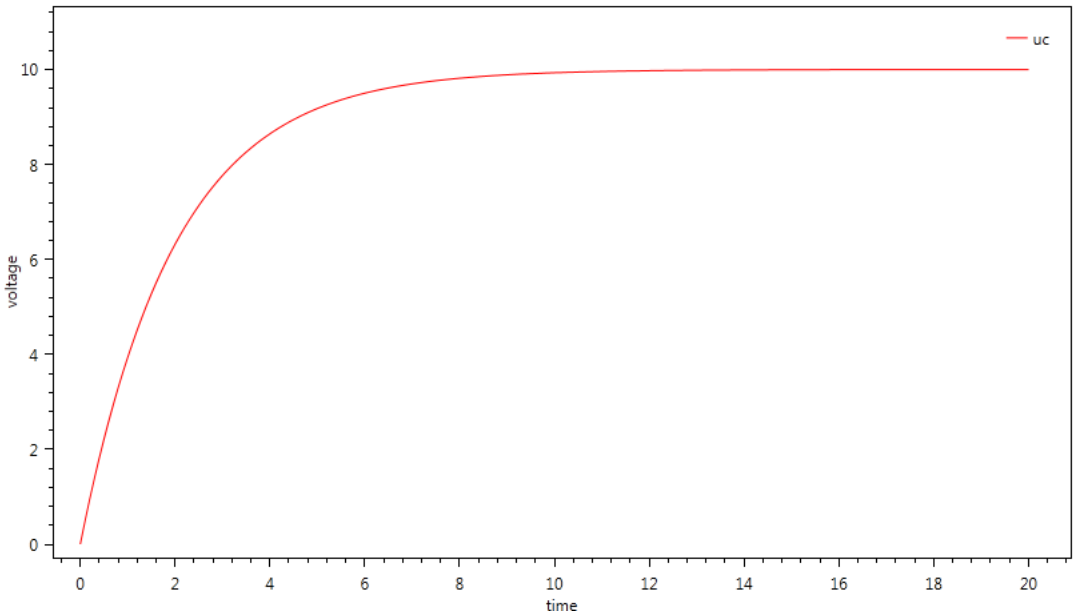
$$\begin{aligned} u'_C &= \frac{U - u_c}{R \cdot C} \\ u_C(0) &= 0 \end{aligned}$$

kde $U = 10V$, $R = 200k\Omega$ a $C = 10\mu F$. Parametry simulace zvolíme následovně – krok $h = 0.01$, celkový čas $t_{max} = 20$ a metodu řešení Runge-Kutta 4. řádu.

Schéma elektrického zapojení vidíme na 5.3, odpovídající blokové schéma je možno nalézt na 5.4. Výsledky (obrázek přímo z aplikace) je označen číslem 5.5.



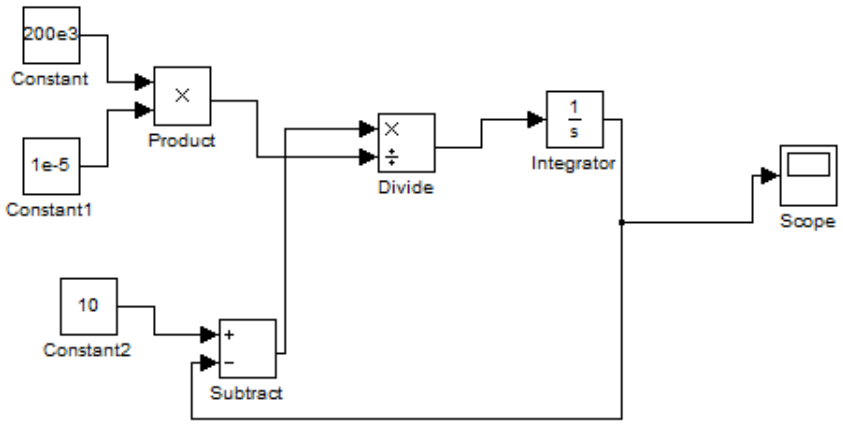
Obrázek 5.4: Blokové schéma vytvořené podle zadání



Obrázek 5.5: Grafický výstup

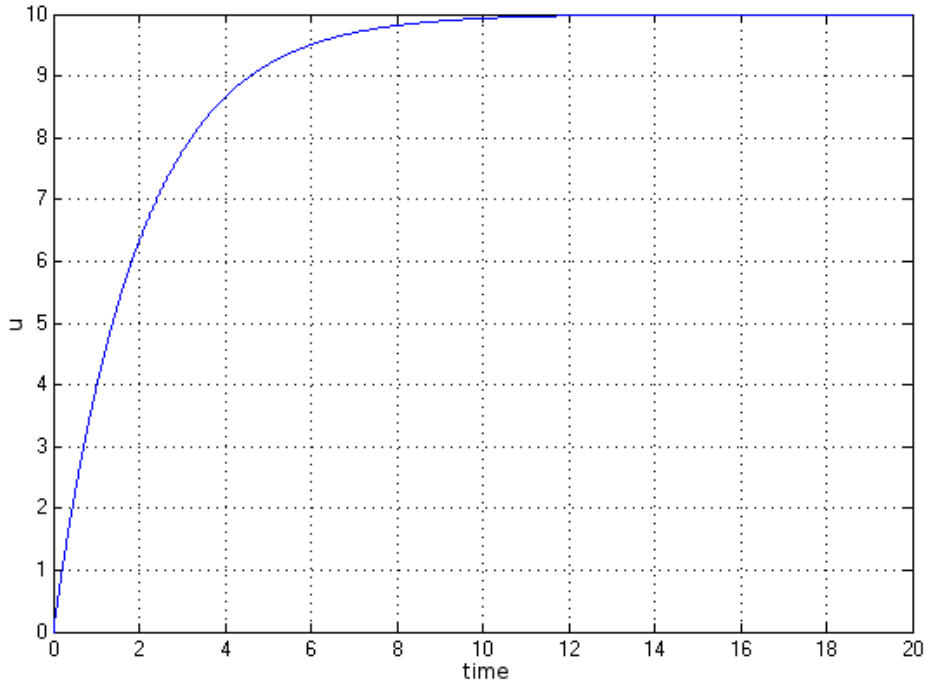
Srovnání se Simulinkem

Při stejném zadání příkladu sestavíme blokové schéma



Obrázek 5.6: Schéma příkladu

nastavíme stejné parametry simulace – krok $h = 0.01$, celkový čas $t_{max} = 20$ a solver ODE4 (odpovídá RK4). Dostáváme následující výsledek[1] – obr. 5.7.



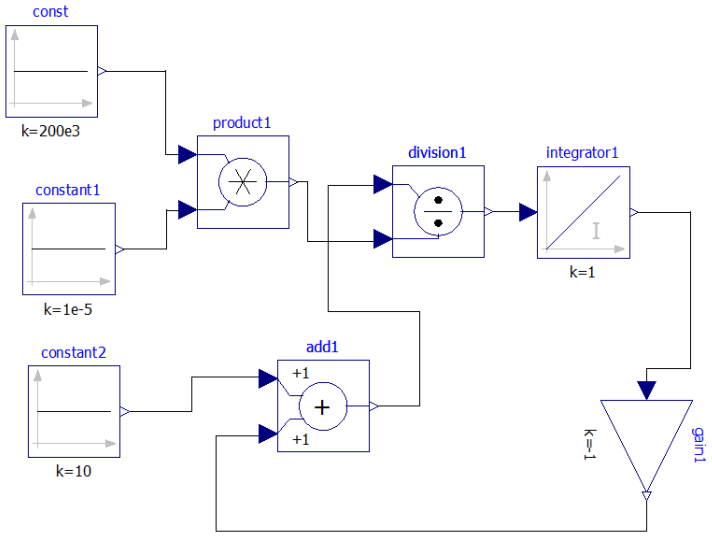
Obrázek 5.7: Výstup z aplikace Simulink

Když vizuálně srovnáme grafy 5.5 a 5.7 tak můžeme říct, že přestože naše aplikace je velmi jednoduchá ve srovnání se Simulinkem, tak na základní úlohy postačuje.

[1] Obrázek byl invertován, některé barvy byly vypuštěny a přidal jsem popisky os, je to ale pořád výstup ze "Scope".

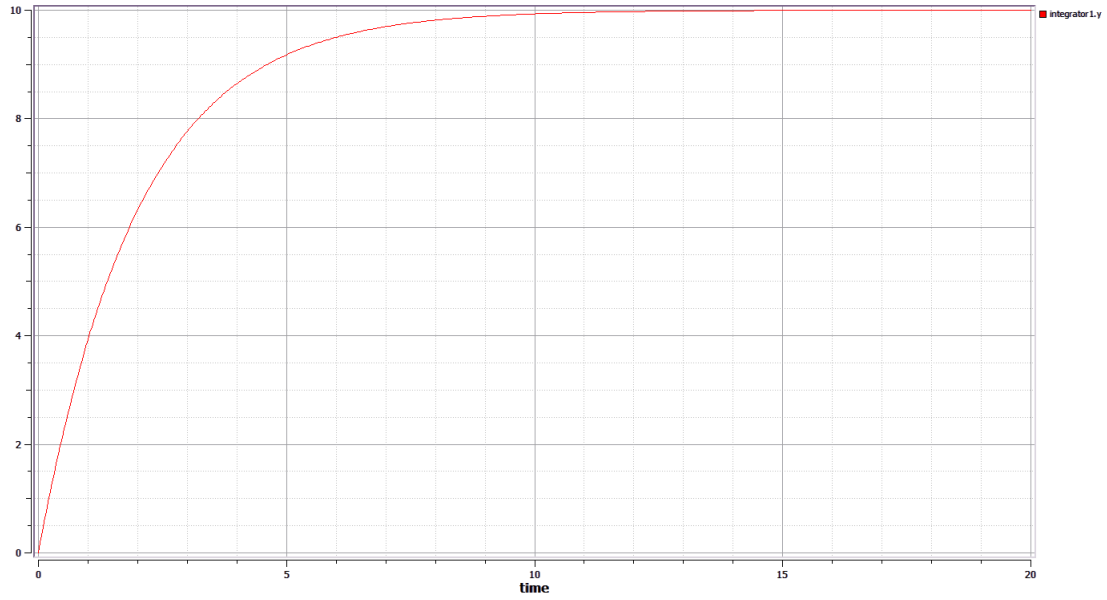
Srovnání s OpenModelicou

Použijeme stejné zadání příkladu, dostaneme následující blokové schéma:



Obrázek 5.8: Schéma příkladu

Nastavil jsem parametry simulace[2] a spustil ji, výstup vidíte na obr. 5.9[3].

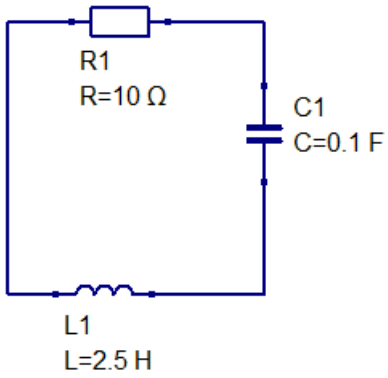


Obrázek 5.9: Výstup z aplikace OpenModelica

Opět můžeme srovnat s naším řešením (obr. 5.5).

[2] Jako solver jsem zvolil "rungekutta".
[3] Barvy opět upraveny, aby graf sedl k okolnímu textu.

5.2.2 RLC obvod – vybíjení



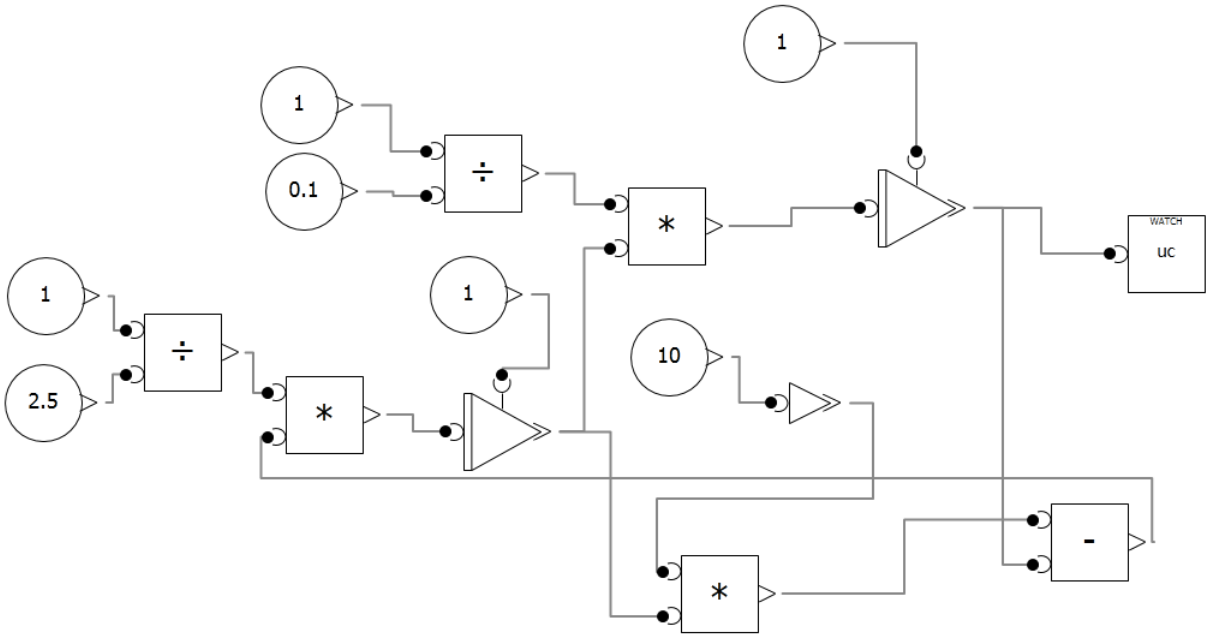
Obrázek 5.10: Obvod RLC (vybíjení)

Řešme následující příklad

$$\begin{aligned} u'_C &= \frac{1}{C} \cdot i_L,\ u_C(0) = 1 \\ i'_L &= \frac{1}{L} \cdot u_L,\ i_L(0) = 1 \\ u_L &= -R \cdot i_L - u_C \end{aligned}$$

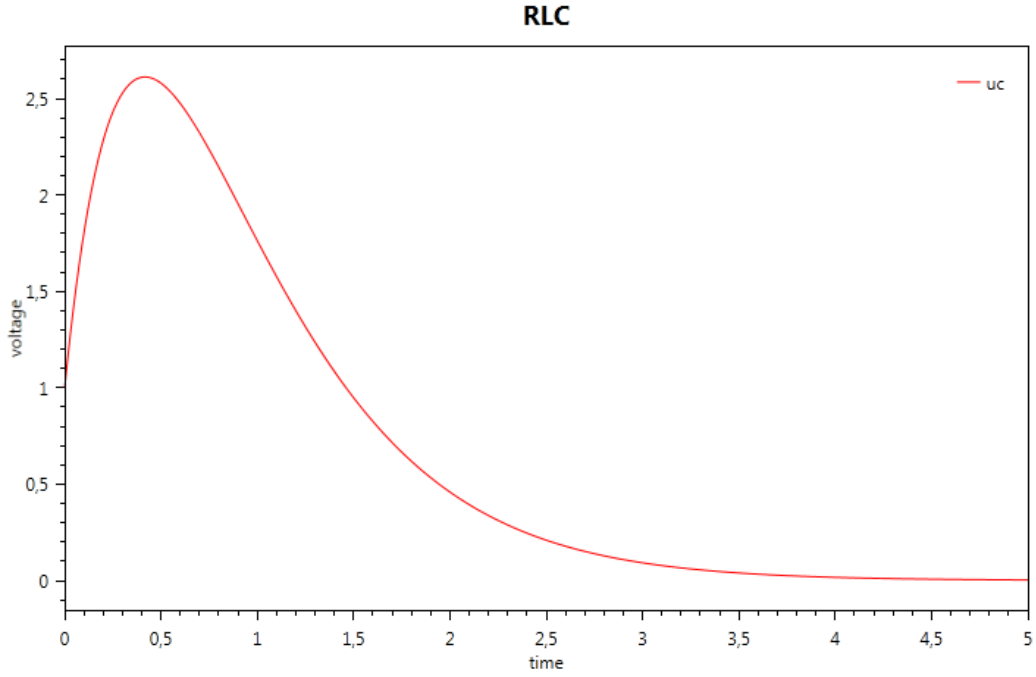
kde $R = 10\Omega$, $L = 2.5H$ a $C = 0.1F$.

V programu jsme vytvořili následující schéma – obr. 5.11.



Obrázek 5.11: Bloková podoba úlohy

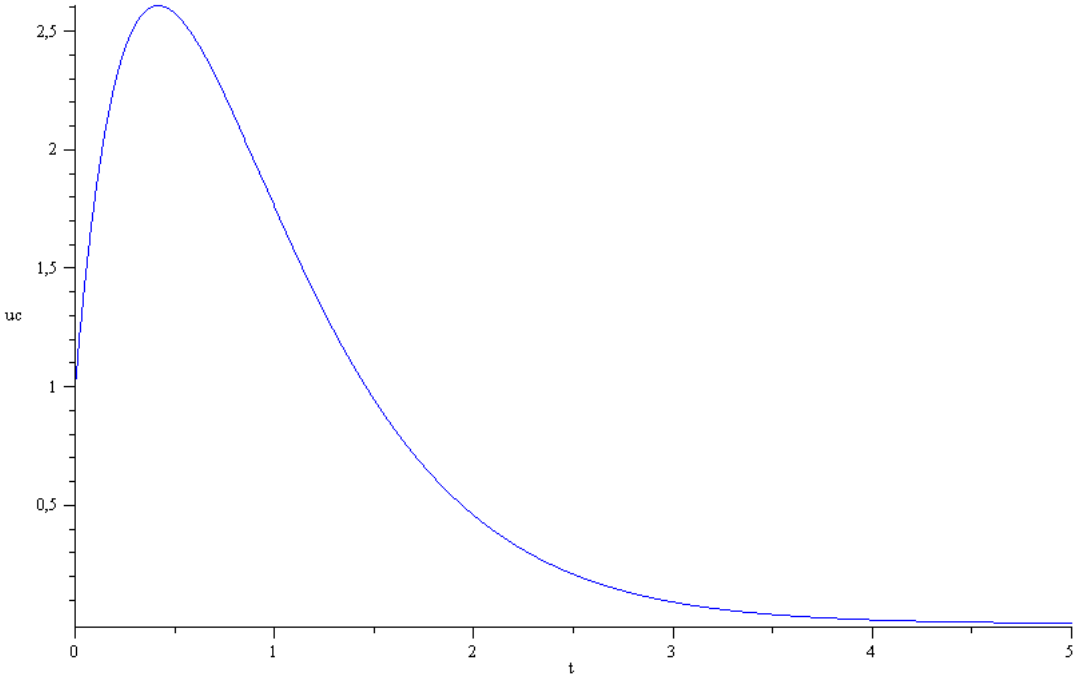
Vhodně jsme zvolili parametry simulace – $h = 0.001$, $t_{max} = 5$. Po přeložení a spuštění simulace dostaneme výstup stejný jako na obrázku 5.12.



Obrázek 5.12: Řešení úlohy RLC vybíjení

Srovnání s aplikací Maple

Po zadání soustavy dostáváme řešení zobrazené na obrázku 5.13. Porovnáme s grafem z naší



Obrázek 5.13: Řešení úlohy RLC vybíjení systémem Maple

aplikace (obrázek 5.12) a můžeme konstatovat, že naše řešení je správné.

Kapitola 6

Závěr

Jedním z cílů bakalářské práce bylo vytvořit aplikaci, která bude schopna řešit soustavy diferenciálních rovnic prvního řádu různými metodami. Tuto část odráží hlavní knihovna, která řeší: překlad vstupu, úpravu získaných dat do standardizované podoby, vygenerování simulační třídy, spuštění simulace, sběr výsledků a jejich výstup. Aplikace umí řešit soustavy několika jednokrokovými metodami s užitím pevného kroku.

Mezikód a následně všechny třídy od sémantické kontroly až po samotnou simulaci podporují všechny základní aritmetické operace včetně umocnění. Oproti TKSL/386 lze vytvářet signál závislý na simulačním čase užitím logických a relačních operátorů a podmíněného výrazu (jsou zde určitá omezení, popsaná v podkapitole 4.2.4). Také jsou podporovány vestavěné funkce, např. *sin*, *abs*, *exp*.

Dalším cílem bylo implementovat grafické rozhraní pro blokový vstup. Po zvážení různých kombinací vzhledů vstupů a výstupů bloku, podob různých operací, jsem nakonec dospěl k finálnímu vzhledu, který jste si mohli prohlédnou např. v podkapitole 4.2. Celkový vzhled blokového editoru (a celé aplikace) byl popsán a předveden v podkapitole 4.13.

Aplikace a její blokový vstup byly úspěšně testovány na základních úlohách, řešení různými numerickými metodami dosahuje předpokládané přesnosti výsledků.

6.1 Budoucí rozšíření

Několik plánovaných rozšíření jsem již zmínil v podkapitole 4.5.5.

Blokový vstup je oproti textovému velmi neúsporný při využití místa na monitoru, proto jako jedno z prvních rozšíření blokového editoru plánuji dokončení podpory pro uživatelem definované bloky. Z dalších plánovaných rozšíření editoru vyjmenuji alespoň několik: více možností pro toolbox (např. zapnutí zobrazení jmen bloků, nyní se zobrazuje pouze jako tooltip), zdokonalit možnosti propojování bloků (např. vytváření "křižovatek", možnost vést spojení po uživatelem specifikovaných bodech), přidat možnost otáčení a zrcadlení bloku, podpora pro pojmenované konstanty (v editoru přibude panel s tabulkou globálních a lokálních konstant, musí se upravit datová vrstva a blokový parser).

Další plánovaná rozšíření se týkají obecných vlastností aplikace, ku příkladu se vážně přemýšlí o implementaci polí (sběrnic), pokročilejších numerických metod (vícekrokových, popř. i metody TKSL) nebo výpočtu `watch` pomocí dynamicky vytvářené třídy (v podstatě jde o stejný princip, kterým je nyní řešena simulace).

Literatura

[1] Diblík, J. a Přibyl, O. *Matematika II: Obyčejné diferenciální rovnice 1.* 2004. Skripta VUT Brno.

[2] Čermák, L. *Numerické metody II.* Brno: CERM, 2004. ISBN 80-214-2722-1.

[3] Fajmon, B. a Růžičková, I. *Matematika 3.* 2006. Skripta VUT Brno.

[4] Hozman, J. *Slajdy pro předmět M3A.* 2011. FP TUL.

[5] Hrubý, M. *Modelování a simulace - spojité modelování.* 2005. Slajdy pro předmět IMS, FIT VUT v Brně.

[6] Kalová. *Dynamické modely.* Plzeň: KMA ZCU, 2007. Diplomová práce.

[7] Konvalina, J. *Datové vstupy a výstupy specializovaného výpočetního systému založeného na numerické integraci.* Brno: FIT VUT v Brně, 2006. Diplomová práce.

[8] Kopecký, J. *Numerický integrátor na platformě .NET.* Brno: FIT VUT v Brně, 2012. Bakalářská práce.

[9] Kubíček, M., Dubcová, M. a Janovská, D. *Numerické metody a algoritmy.* Praha: VŠCHT Praha, 2005. ISBN 80-7080-558-7.

[10] Maplesoft. *Maple User Manual.* Canada: [b.n.], 2012. ISBN 978-1-926902-23-4.

[11] Meduna, A. a Lukáš, R. *Formální jazyky a překladače.* 2011. Slajdy pro předmět IFJ, FIT VUT v Brně.

[12] Microsoft. *MSDN Library* [online]. 2012. Dostupné na: <http://msdn.microsoft.com/en-us/library>.

[13] Peringer, P. *Modelování a simulace.* 2011. Slajdy pro předmět IMS, FIT VUT v Brně.

[14] Vach, M. *Numerické řešení DR - Runge-Kutta metoda, Galerkinova metoda (okrajové úlohy).* Slajdy pro předmět Fyzikálně - chemické aspekty procesů v prostředí, KVHEM CZU.

[15] Vitásek, E. *Numerické metody.* Praha: SNTL, 1987.

[16] *Laboratorní přípravek IPR* [online]. 2010. Dostupné na: <http://www.fit.vutbr.cz/~satek/FRVS/index.php>.

Příloha A

Gramatika jazyka

```
/* Struktura programu */
<program> → <program-parts> <sysblock> <program-parts>

<sysblock> → system { <block-body> }
<sysblock> → ε

<program-parts> → ε
<program-parts> → <program-part> <program-parts>

<program-part> → <const-stat>;
<program-part> → <block-definition>


/* Definice uživatelského bloku */
<block-definition> → block <block-name>(<block-params>) { <block-body> }

<block-name> → <id>

<block-params> → <block-params-list>

<block-params-list> → <block-params-type> <id> <block-params-list-cont>
<block-params-list-cont> → ε
<block-params-list-cont> → , <block-params-list>

<block-params-type> → in
<block-params-type> → out

<block-body> → <statments-sequention>

/* Příkazy */
<statments-sequention> → <statment-with-semicolon><statments-sequention>
<statments-sequention> → ε
```

```
<statment-with-semicolon> → <statment>;
<statment-with-semicolon> → <block>
<statment-with-semicolon> → <for>

<statment> → <equation-or-block-call-or-unop>
<statment> → <var-stat>
<statment> → <const-stat>
<statement> → <watch-stat>

<for> → for ( <for-variables> ; <const> ; <for-variables> ) <statment-with-semicolon>
<for-variables> → ε
<for-variables> → <for-variables-one>
<for-variables-one> → <id> <for-variables-list>
<for-variables-list> → , <for-variables-one> <for-variables-list>
<for-variables-list> → ε

<foreach> → foreach ( var <id> in <id(type=array)> ) <statment-with-semicolon>

<block> → { <statments-sequention> }


/* Definice promenných */
<var-stat> → var <var-stat-one-var> <var-stat-list>
<var-stat-one-var> → <id> <var-stat-set-val-opt>
<var-stat-set-val-opt> → ε
<var-stat-set-val-opt> → = <expression>
<var-stat-list> → ε
<var-stat-list> → , <var-stat-one-var> <var-stat-list>

/* Definice konstant */
<const-stat> → const <const-stat-one-const> <const-stat-list>
<const-stat-one-const> → <id> = <expression(type=const)>
<const-stat-list> → ε
<const-stat-list> → , <const-stat-one-const> <const-stat-list>

/* Definice watches */
<watch-stat> → watch <watch-stat-one-watch> <watch-stat-list>
<watch-stat-one-watch> → <id> <watch-stat-set-val-opt>
<watch-stat-set-val-opt> → ε
<watch-stat-set-val-opt> → = <expression>
<watch-stat-list> → ε
<watch-stat-list> → , <watch-stat-one-watch> <watch-stat-list>

/* Začátek přiřazení/integrace, nebo volání bloku */
<equation-or-block-call-or-unop> → <id> <equation-or-block-call-or-unop-sec-part>
<equation-or-block-call-or-unop-sec-part> → <block-call>
<equation-or-block-call-or-unop-sec-part> → <equation>
```

```
/* Volání bloku */
<block-call> → (<block-call-params>)
<block-call-params> → <block-call-params-list>
<block-call-params-list> → <block-call-params-list-one-arg> <block-call-params-list-cont>
<block-call-params-list-one-arg> → <expression>
<block-call-params-list-one-arg> → out <id>
<block-call-params-list-cont> → ε
<block-call-params-list-cont> → , <block-call-params-list>

/* Přiřazení/integrace */
<equation> → = <expression>
<equation> → ' = <expression> & <expression(type=const)>


/* Výrazy */
/* <expression>! LL */
<expression> → <id>
<expression> → <const>
<expression> → ( <expression> )
<expression> → <operator-un-pre> <expression>
<expression> → <expresion> <operator-bin> <expression>
<expression> → <expression(type=bool)> ? <expression> : <expression>
<expression> → <built-in-function>
<expression> → <block-call>
<expression> → <operator-un-var>
<expression> → <expression> in <expression>..<expression>

<operator-bin> → <operator-bin-ar> | <operator-bin-log>
<operator-bin-ar> → + | - | * | / | ^
<operator-bin-log> → && | || | and | or
<operator-bin-log> →<|>|<= |>= | == | !=
<operator-un-pre> → <operator-un-pre-ar> | <operator-un-pre-log>
<operator-un-pre-ar> → -
<operator-un-pre-log> → ! | not

/* Číselná konstanta */
<const> → <const-dec-number><const-e-opt>
<const-dec-number> → <const-minus-opt><num-rep><const-dec-part-opt>
<const-dec-part-opt> → ε
<const-dec-part-opt> → .<num-rep>
<const-minus-opt> → ε
<const-minus-opt> → -
<const-e-opt> → ε
<const-e-opt> → (e|E)<const-minus-opt><num-rep>
```

```
<num-rep> → <num><num-rep-cont>
<num-rep-cont> → ε
<num-rep-cont> → <num><num-rep-cont>

/* Identifikátor */
<id> → <char-or-under> <id-cont-opt>
<id-cont-opt> → ε
<id-cont-opt> → <num-or-char-or-under> <id-cont-opt>
<num-or-char-or-under> → <num-or-char> | _
<num-or-char> → <num> | <char>
<char-or-under> → <char> | _
<num> → 0..9
<char> → a..z | A..Z



/* Gramatika pro precedencni analyzu */
/* Vyrazy */

/* bez priority */
<E> → i
<E> → c
<E> → ( <E> )
<E> → s
<L> → <P> , <L>
<L> → <P>
<P> → out i
<P> → <E>

/* odsud prioritni skupiny. cim vys, tim vyssi priorita */

<E> → i( <L> )

<E> → - <E>
<E> → ! <E>

<E> → <E> ^ <E>

<E> → <E> * <E>
<E> → <E> / <E>

<E> → <E> + <E>
<E> → <E> - <E>
```

<E> → <E><<E>
<E> → <E>><E>
<E> → <E><= <E>
<E> → <E>>= <E>

<E> → <E> == <E>
<E> → <E> != <E>

<E> → <E> and <E>
<E> → <E> or <E>

<E> → <E> ? <E> : <E>

Příloha B

Použité knihovny

- OxyPlot – komponenta vykreslující graf
- Ninject – knihovna starající se o depedency injection
- AvalonDock – panely v GUI
- WPF Diagram Designer – použit jako podklad pro editor schémat, většina částí byla přepsána
- AvalonEdit – textový editor
- moq – užívá se při testování pro "mockování"
- xUnit.net – použit pro unit testy (skoro všechny třídy v Core mají svoje sady testů)

Příloha C

Obsah přiloženého média

- src – zdrojové soubory aplikace
- bin – přeložená aplikace (ke spuštění je nutné mít nainstalován .NET 4.0)
- tex – tato práce, ostatní dokumenty